Reinforcement Learning-Assisted Virtualized Security Framework for CAVs.

Final Report

by

Jagruti Sahoo
South Carolina State University

Arthur Noa Mukwaya, South Carolina State University
Denis Ruganuza, South Carolina State University
Izison Benibo, South Carolina State University
Nana Kankam Gyimah, South Carolina State University
Judith Mwakalonge, South Carolina State University
Gurcan Comert, North Carolina Agricultural and Technical State University
Biswajit Biswal, South Carolina State University
Nikunja Swain, South Carolina State University
Yanzhou Fu, Benedict College

June 2025

NATIONAL CENTER FOR TRANSPORTATION CYBERSECURITY AND RESILIENCY (TraCR)

DISCLAIMER

The contents of this report reflect the views of the authors, who are responsible for the facts and the accuracy of the information presented herein. This document is disseminated in the interest of information exchange. The report is funded, partially or entirely, by the National Center for Transportation Cybersecurity and Resiliency (TraCR) under Grant No. 69A3552344812 and 69A3552348317 which is headquartered at Clemson University, South Carolina, USA, from the U.S. Department of Transportation's University Transportation Centers Program. The U.S. Government assumes no liability for the contents or use thereof.

CONTACTS

For more information:

TraCR
Clemson University
One Research Dr
Greenville, SC 29607
tracr@clemson.edu

Jagruti Sahoo
South Carolina State University
300 College Street, NE
Orangeburg, SC 29117
Phone: 803-536-8968
Email: jsahoo@scsu.edu

Judith Mwakalonge
South Carolina State University
300 College Street, NE
Orangeburg, SC 29117
Phone: 803-536-8321
Email: jmwakalo@scsu.edu

ACKNOWLEDGMENT

This work is based upon the work supported by the National Center for Transportation Cybersecurity and Resiliency (TraCR) (a U.S. Department of Transportation National University Transportation Center) headquartered at Clemson University, Clemson, South Carolina, USA. Any opinions, findings, conclusions, and recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of TraCR, and the U.S. Government assumes no liability for the contents or use thereof.

Technical Report Documentation Page

1. Report No. 12	**2. Government Accession No.** N/A	**3. Recipient's Catalog No.** N/A
4. Title and Subtitle Reinforcement Learning-Assisted Virtualized Security Framework for CAVs.		**5. Report Date:** June 2025 **6. Performing Organization Code:** N/A
7. Author(s) Jagruti Sahoo, Ph.D. https://orcid.org/0000-0002-6616-0926 Arthur Noa Mukwaya, https://orcid.org/0009-0007-1987-3134 Denis Ruganuza, https://orcid.org/0009-0005-7701-6072 Izison Benibo, https://orcid.org/0009-0006-1737-3905 Nana Kankam Gyimah, Ph.D. https://orcid.org/0009-0008-0601-440X Judith Mwakalonge, Ph.D. https://orcid.org/0000-0002-7497-6829 Gurcan Comert, Ph.D. https://orcid.org/0000-0002-2373-5013 Biswajit Biswal, Ph.D. https://orcid.org/0000-0002-7287-9419 Nikunja Swain, Ph.D. https://orcid.org/0009-0007-6740-5861 Yanzhou Fu, Ph.D. https://orcid.org/0000-0002-5471-9972		**8. Performing Organization Report No.** 12
9. Performing Organization Name and Address National Center for Transportation Cybersecurity and Resiliency (TraCR), Clemson University, 414 A One Research Dr, Greenville, SC 29607 South Carolina State University 300 College Street, NE Orangeburg, SC 29117		**10. Work Unit No.** N/A **11. Contract or Grant No.** 69A3552344812 and 69A3552348317
12. Sponsoring Agency Name and Address U.S. Department of Transportation Office of the Assistant Secretary for Research and Technology 1200 New Jersey Avenue, SE, Washington, DC 20590		**13. Type of Report and Period Covered** Final Report, 01/01/2024 - 05/31/2025 **14. Sponsoring Agency Code** OST-R
15. Supplementary Notes Conducted under the U.S. DOT Office of the Assistant Secretary for Research and Technology's (OST-R) University Transportation Centers (UTC) program.		
16. Abstract Security vulnerabilities in the Connected and Autonomous Vehicle (CAV) software can allow hackers to perform malicious actions ranging from draining batteries and taking control of the steering wheel to disabling the alarm system. CAV software is subject to several cyber-attacks including memory corruption, code injection, remote code execution, and malware attacks. This study develops a Network Functions Virtualization (NFV)-based virtualized security framework to increase the resilience of CAV software. The proposed framework integrates a reinforcement learning-based code diversification mechanism that employs a Double Deep Q Network (DDQN) agent to optimally execute code variants of CAV software implemented as virtual network functions (VNFs). By meticulously switching the VNFs, the framework presents an unpredictable attack surface, thereby ensuring resilience. Our results show that the DDQN agent achieves a higher security access rate and higher uptime than a Q-learning agent, demonstrating the ability of the DDQN agent to provide an improved defense response and maintain continuity in the event of intrusions. Our prototype demonstrates the feasibility of hosting the CAV software programs as VNFs on a virtualized infrastructure. Moreover, we demonstrate the benefit of code diversification by switching the implementation of a point cloud concatenation functionality across different languages and simulating a memory corruption attack to test the program behavior. The research methods developed by this study can be applied to realize optimal software diversification in other areas such as the Internet of Things (IoT) and Cyber-physical Systems (CPS).		
17. Keywords Buffer Overflow, Connected and Autonomous Vehicles (CAVs), Cybersecurity, Code Diversification, Moving Target Defense (MTD), Reinforcement Learning, Network Function Virtualization (NFV)		**18. Distribution Statement** No restrictions.

19. Security Classif. (of this report) Unclassified	**20. Security Classif. (of this page)** Unclassified	**21. No. of Pages** 69	**22. Price** N/A

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

TABLE OF CONTENTS

List of Tables

List of Figures

EXECUTIVE SUMMARY

Over the last few years, there has been significant growth in automotive software. It plays an integral role in Connected and autonomous vehicles (CAV), where critical functionalities such as perception, decision-making, and control are operated by software. However, security vulnerabilities in the CAV software allow hackers to compromise critical modules and perform malicious actions ranging from draining batteries and taking control of the steering wheel, to disabling the alarm system. CAV software is subject to several cyber-attacks including memory corruption, code injection, remote code execution, and malware attacks. Securing CAV software is therefore of paramount importance to ensure the safe and reliable operation of CAVs.

Code diversification is a viable approach to improving the resiliency of CAV software by meticulously altering the code. This study develops a robust code diversification-based defense mechanism that leverages language diversity to execute multiple variants of a functionally equivalent code. To efficiently manage the code variants and facilitate their dynamic execution, this study develops a virtualized security framework designed based on the Network Functions Virtualization (NFV) paradigm. The proposed framework can scale as needed to support the accelerated growth in CAV software and allows CAV software to be implemented as Virtual Network Functions (VNFs). The framework integrates the code diversification mechanism and follows the principle of moving target defense to introduce uncertainty in the attack surface, making it harder for hackers to exploit vulnerabilities.

It is a critical challenge to select the optimal code variants due to the diversity of the language-specific vulnerabilities, the dynamic intrusion behavior of hackers, and an incurred cost associated with maintaining the VNFs representing the code variants. This study addresses this challenge by developing reinforcement learning (RL)-based methods (Q-learning and Double deep Q Network (DDQN)) that allow an agent to learn the optimal policy (i.e., the optimal code variant that maximizes resiliency, maximizes QoS (Quality of service), and minimizes the cost of resource consumed by the code) by interacting with the CAV environment. Our results show that the DDQN consistently receives higher reward Q-Learning over time, indicating better policy convergence. Our DDQN agent achieves a QoS (i.e., uptime) of 98.66%, thereby confirming its ability to ensure the continuity of CAV services even in the presence of intrusions. The Q-learning agent achieves a slightly lower QoS (97.62%). We also observe that DDQN achieves a higher mean time between intrusions compared to Q-learning, proving the ability of DDQN to ensure resilience. The DDQN agent maintains a higher security success rate than Q-Learning. It demonstrates the ability of the DDQN agent to provide an improved defense response in the event of intrusions.

This study also investigates a buffer overflow scenario and demonstrates the possibility of corrupting memory during the execution of point cloud concatenation functionality. This study develops a prototype of the proposed framework using the OpenStack platform and evaluates the code diversification performance by executing multiple VNFs of the point cloud concatenation functionality. We demonstrate that switching from one VNF to another can prevent hackers from exploiting language-specific behaviors, thereby showing the effectiveness of the proposed virtualized framework in ensuring resilience.

Future work will include the development of a modeling RL agent using game theoretical models to capture the actions of an adversary during training. Moreover, the multi-agent RL method can be investigated to allow cooperative and competitive agents to handle multi-point cyberattacks across different CAV subsystems (e.g., Perception, Navigation, Diagnostics) simultaneously.

CHAPTER 1

Introduction

1.1 Background and Motivation

Connected and autonomous vehicles (CAV) have brought a major transformation in the transportation sector by significantly improving the mobility of people and goods through advanced communication, sensing, and computing capabilities. CAVs depend on software that processes input from sensors including lidar, radar, and camera, and execute complex algorithms to provide critical functions such as perception, decision-making, and control. The software used in CAVs is ever-growing both in terms of size and complexity due to increased integration of autonomous driving and adoption of Artificial Intelligence (AI) technology for real-time decision making. According to Goldman Sachs Research, the lines of code per vehicle doubled in five years from 2015 to 2020 reaching 200 million. Moreover, the CAV software will grow further, and each car will be equipped with 650 million lines of code by 2025 (Goldman Sachs, 2022).

An increased attack surface as a result of the continued growth will result in increased cybersecurity risks for CAVs, thereby jeopardizing their safe and reliable operation. Typical attack vectors such as OBD-II ports, USB ports, infotainment systems, wireless interfaces (e.g., Wifi and Bluetooth), and OTA (Over-the-air) updates allow hackers to gain access to ECUs and exploit vulnerabilities in the ECU software (Elkhail et al., 2021). CAV software is subject to several cyber-attacks including memory corruption, code injection, remote code execution, and malware attacks. [reference]. Security researchers have demonstrated various incidents of compromising vulnerable CAV software that involved updating the firmware (Eiza & Ni, 2017), remotely unlocking and starting the vehicle (Muncaster, 2025), disabling brakes and taking control over the steering wheel (Eiza & Ni, 2017).

Although CAV software undergoes various types of testing including static analysis, dynamic analysis, fuzz testing, and penetration testing before its deployment, several factors are responsible for making it vulnerable once the software gets deployed and starts working on a live vehicle. First, insufficient testing may leave bugs and logic errors unattended. 2) A piece of code that has passed the test may become exploitable under new and evolving attack surfaces, 3) Costly update procedures leave the CAV software unpatched for a long period of time

Code diversification (Potteiger et al., 2022) (Hosseinzadeh et al., 2018). is a viable strategy that can certainly provide protection to CAV software during runtime by allowing the execution of multiple variants of a functionally equivalent code. By using the principles of Moving Target Defense (MTD) (Cho et al., 2020), the code diversification mechanism can introduce uncertainty in the attack surface of the CAV software. MTD involves changing the attack surface at fixed intervals to increase the complexity of the system and make it difficult for hackers to discover and exploit vulnerabilities (Potteiger et al., 2022) (Cho et al., 2020).

Although code diversification has been explored in traditional IT domains, its application to the automotive domain is still limited (Potteiger et al., 2022). Only a handful of works (Yi et al., 2020) (Hataba et al., 2022) (Hataba et al., 2021) explored code diversity in the CAV domain. Hataba et al., (2022) proposed a compiler-based diversification technique to secure code execution in a vehicular cloud computing environment. Since this approach is designed for securing the code requested by remote users (e.g., smartphone users), it may not be directly applicable to CAV software. Yi et al. (2020) presented a control-flow flattening approach to obfuscate code used on embedded systems. Their approach is designed for only

C based software and therefore may not be applicable to secure a diverse automotive software environment. This study addresses the current gap in the literature by proposing a code diversification mechanism that leverages language diversity in an optimal way to secure CAV software.

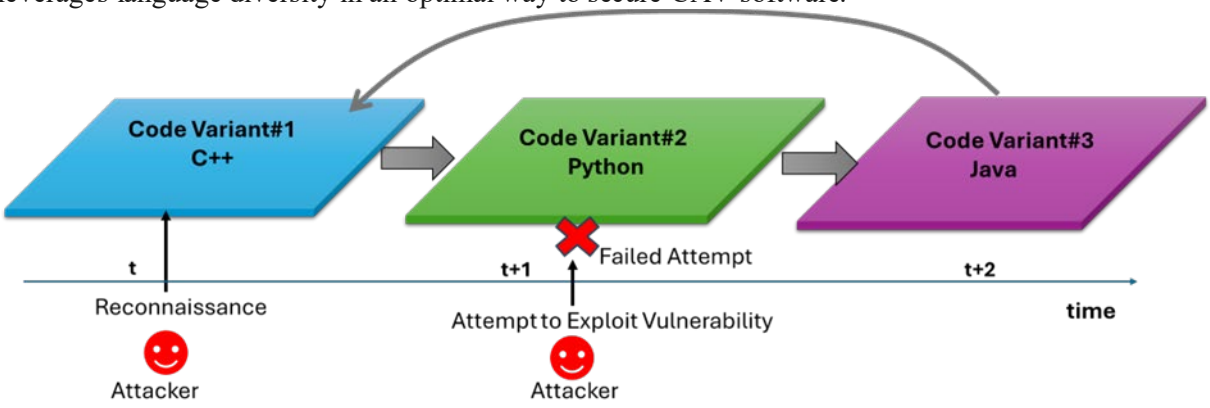

Figure 1: Code Diversification Scenario.

Figure 1 shows a code diversification scenario, where code variants of CAV software are executed at a different time instant. As shown in Figure 1, a hacker successfully performs reconnaissance and identifies vulnerabilities in code variant#1 (C++ version) at time t. At t+1, when hackers get ready to exploit the vulnerabilities, those C++-specific vulnerabilities can no longer be exploited, as the code variant at t+1 uses a Python-based implementation. To harness the benefits of MTD, the attack surface must be moved in an optimal way. However, the diversity of the language-specific vulnerabilities, the dynamic intrusion behavior of hackers, and an incurred cost associated with maintaining multiple variants make it challenging to select the optimal code variant. Moreover, there is a need to develop a framework that supports the dynamic execution of the code variants to facilitate code diversification, reduce maintenance efforts, reduce operational costs, and scale as needed to support future growth in CAV modules.

1.2 Objectives

The objectives of this study are to 1) develop a novel virtualized security framework that adopts code diversification-based defense to improve the resiliency of the CAV software, 2) develop a reinforcement learning (RL) based algorithm to dynamically select the optimal code variant, 3) evaluate the performance of the RL algorithm using simulations, and 4) develop a proof-of-concept of the virtualized security framework.

Our virtualized security framework is designed based on the NFV paradigm (Farris et al., 2019) to leverage its inherent benefits including scalability, elasticity, and reduced maintenance and efforts. Our framework implements the code diversification mechanism that allows the execution of multiple code variants of CAV software and switches the code variants dynamically to prevent the hacker from exploiting basic as well as advanced vulnerabilities. Few works have shown the possibility of implementing CAV functionality as VNFs such as a computer vision model running on an edge device (Coronado et al., 2020) and routing protocols running on an in-vehicle NFV platform (Zhu et al., 2016).

Using the proposed framework, the code variants are implemented as virtual network functions (VNFs) or Virtual Functions (VFs) that can run on a virtualized infrastructure, leading to reduced CAPEX and OPEX.

We consider buffer overflow-based use case as it is one of the common vulnerabilities found in C/C++ based software in CAVs. We demonstrate the impact of buffer overflow on the performance of a C++ program responsible for combining point clouds published by lidar sensors.

Reinforcement Learning has demonstrated a significant promise in enabling adaptive decision-making for security-critical systems, including autonomous vehicular platforms (Nguyen & Reddi, 2021). In this study, RL is applied to allow the code diversification mechanism to execute an optimally selected VNF that provides improved resilience to intrusions. We investigated Q-learning (basic RL) and DDQN (advanced RL) to decide the optimal policy.

This study also develops a prototype using the Openstack platform (Openstack, n.d.). The prototype implements VNFs corresponding to CAV software in multiple languages. The prototype also integrates a simplified code diversification mechanism that switches between VNFs to exhibit MTD behavior, thereby improving the resilience of the CAV software.

The remainder of the report is organized as follows. Chapter 2 discusses the related work focusing on code diversification methods and reinforcement learning for security. Chapter 3 presents the methodology including buffer overflow exploitation, the proposed virtualized security framework, the proposed RL-based optimal code diversification mechanism, experimental setup, and performance metrics. Chapter 4 presents the performance evaluation. Chapter 5 outlines the key findings and discusses future work.

CHAPTER 2

Literature Review

2.1 Code Diversification Methods

Some of the works on code diversification include the adaptive method by (Jangda et al., 2015). That method continuously regenerates diversified Java bytecode during program execution, thereby shrinking the window of opportunity for code-reuse attacks like ROP (Return-Oriented Programming). Though that method achieved better performance than prior static or one-time dynamic diversification methods, the approach relies heavily on profiling accuracy, which can vary significantly with different workloads or runtime behaviors. It assumes that source code access is possible at runtime, which may not be viable in all deployment scenarios. Styugin et al (2016) presented code diversity-based strategies for interpreted languages (e.g., Python). Their approach dynamically transforms the source code of interpreted programs, such as creating random variables, and inserting additional code. Taguinod et al. (2015) presented a translator-based technique to change web application's implementation from one language to another. However, using a translator may incur a delay in creating a new version of the application code and hence it may not meet the real-time requirements of CAV services.

To protect Internet Things (IoT) devices from cyberattacks, (Mahmood & Shila, 2016) Proposed a novel MTD framework combining context-aware code partitioning with code diversification where only minimal trusted code resides on the device, and additional code is downloaded from a secure cloud source depending on runtime context. The code is erased after use, and diversified versions of code are delivered each time to prevent attackers from predicting software behavior or exploiting known vulnerabilities. Although the framework increases the difficulty of launching successful cyberattacks, the need to download code from the cloud every time the functionality is requested incurs high communication overhead. Moreover, it uses the same implementation that increases the opportunity for hackers to understand the code structure. In the embedded systems domain, Tsoupidi et al. (2023b) introduced code diversification and interrupt-level randomization method against code-reuse and side channel attacks. The system randomizes code layouts at compile time and dynamically re-randomizes interrupt handlers during runtime to invalidate attacker assumptions about control flow and timing.

To the best of our knowledge, there has been limited work (Hataba et al., 2021b, 2022b) and (Yi et al., 2020) on the application of code diversity in the CAV domain. Those works (Hataba et al., 2021b, 2022b) focus on protecting software running in the AVCC (Autonomous Vehicular Cloud Computing) platforms by using a modified compiler that applies dynamic control-flow obfuscation to generate diversified code to secure remote code execution. Since the diversification approach is designed to secure code of smartphone or small computer users, it may not be directly applicable for CAV software. Yi et al. (2020) presented control-flow flattening approach to obfuscate C code. They performed their diversity experiments on embedded systems typically used in automotive systems. Since their approach is designed for only C based software, it may not be applicable to secure a diverse automotive software environment.

Diversifying code at the language level is still yet to be investigated in the CAV domain. Moreover, majority of the above works (Jangda et al., 2015;.Styugin et al., 2016; Taguinod et al., 2015; Mahmood & Shila, 2016; Hataba et al., 2021b, 2022b) (Yi et al., 2020) did not address the selection of optimal code variants. To address the above gaps, this study develops a virtualized framework that offers a scalable and agile platform to facilitate the execution of variants of CAV software. The variants are obtained by a combination of diversity methods: 1) modify the structure of a program, and 2) generate versions in different languages.

This study also proposes a reinforcement learning based approach to determine the optimal implementation that provides maximum protection to the CAV software.

2.2 Reinforcement Learning for Security

Connected and Autonomous Vehicles (CAVs) represent complex cyber-physical systems that depend heavily on sophisticated software modules to perform critical tasks like perception, navigation, and diagnostics (Parekh et al., 2022). These increasingly sophisticated systems, often operating in real-world environments, are prime targets for advanced cyber threats, including zero-day exploits and memory corruption attacks. Traditional static defense techniques like fixed firewalls, signature-based intrusion detection systems, and routing patching lack the adaptive ability that is required to withstand these dynamic and evolving threats. Their static nature renders them inefficient against novel attack vectors, leaving CAVs exposed (Rana & Hossain, 2023).

Reinforcement Learning (RL) has emerged as a promising paradigm for cybersecurity due to its adaptability and feedback-driven defense learning capabilities in complex and evolving environments (Andreou et al., 2025). RL agents continuously refine their strategies through real-time interaction with the environment, enabling proactive, context-aware security responses (Jadhav et al., n.d.). (Nguyen & Reddi, 2023)provided a comprehensive survey of RL-based cyber defense agents, demonstrating their effectiveness in advanced intrusion detection systems (IDS), phishing prevention, and denial-of-service (DoS) mitigation, highlighting RL's versatility across multiple domains. In industrial cyber-physical systems, (Cengiz & Gök, 2023) validated the applicability of RL—specifically Deep Q-Network (DQN) and Partially Observable Markov Decision Processes (POMDP)—to real-world stealthy attack detection. Within the vehicular domain,(Khlifi et al., 2025) successfully implemented a Double Deep Q-Network (DDQN) integrated into a semantic perception pipeline, to improve decision making for real-time autonomous systems, emphasizing DDQN's effectiveness in managing high-dimensional, dynamic state spaces.

While vanilla DQNs have proven to be powerful, they are known to suffer from overestimation bias and exhibit learning instability, especially in adversarial and dynamic environments (van Hasselt et al., 2015). This bias stems from the max operation used in Q-learning, which can unintentionally select overestimated Q-values, leading to suboptimal policies. DDQN explicitly addresses these limitations by decoupling the action selection process from the value estimation process. It uses a Q-network for selecting actions in the next state and a separate target network to estimate the value of those actions (Liu et al., 2025).

CHAPTER 3

Methods

3.1 Buffer Overflow in CAV Software

Buffer overflow is a common vulnerability found in software written in C/C++. Since most of the critical CAV modules are also developed as C/C++ programs, we focused on the buffer overflow vulnerability and conducted experiments to demonstrate its real-time impact such as corruption of memory and crashing of a CAV software program.

3.1.1 Exploitation Workflow

Our exploitation workflow is shown in Figure 2. It consists of simulation-based emulation of real-time CAV behavior using AWSIM (*AWSIM Document*, n.d.) , an open-source traffic simulator, with Autoware (The Autoware Foundation GitHub, n.d.), static code analysis, design and execution of a buffer overflow exploit targeting Lidar point cloud concatenating node and monitoring of the attack using the GDB tool. The Autoware stack has 5 modules for sensing, perception, localization, planning, and control. It processes sensor inputs such as LiDAR point clouds simulated by AWSIM, which create realistic traffic scenarios. We conducted static analysis using Flaw Finder tool to identify vulnerabilities such as memcpy operations without bounds checking and vsnprint operation for format string exploitation in the CAV C++ software programs. The output of the flaw finder tool is provided in APPENDIX F.

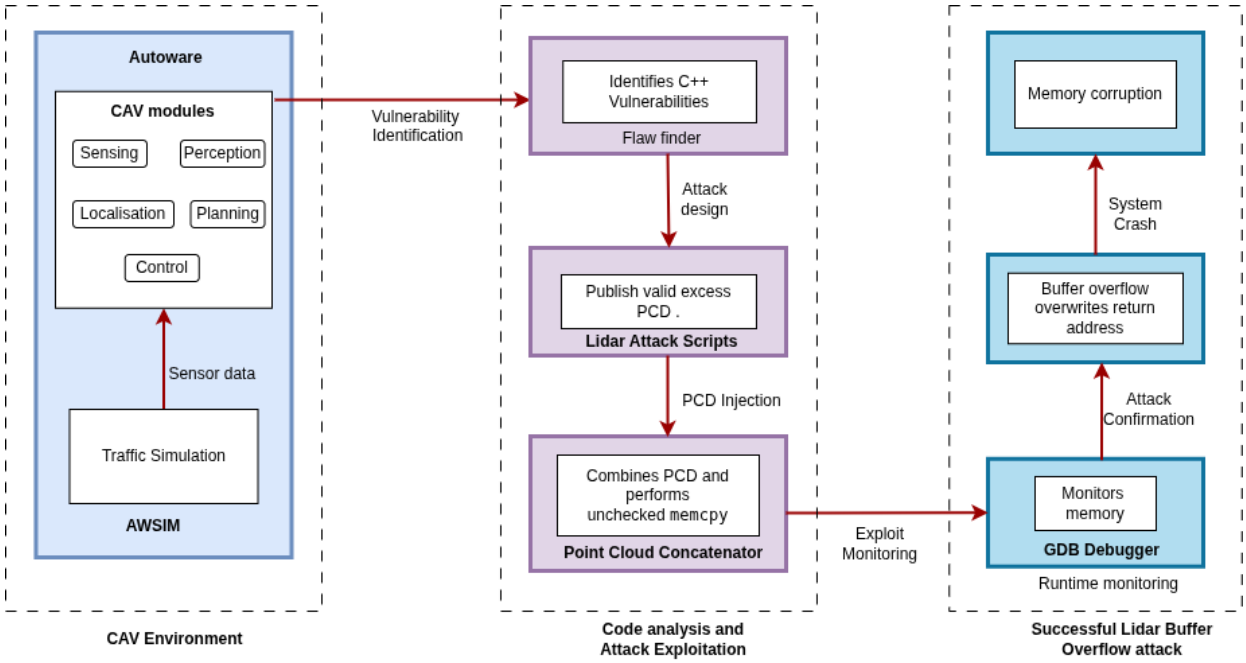

Figure 2: Workflow for Buffer Overflow Exploitation (PCD: Point Cloud Data)

3.1.2 Exploitation of memory operation

We selected the memcpy() operation to trigger a buffer overflow attack in a vulnerable C++ program representing a ROS node that is responsible for concatenating point cloud data received from lidar sensors. The memcpy() function is used by the ROS node to write the individual point cloud data into adjacent regions in memory. Under attacking scenario, we assume that a hacker acts as publishers and send excessive point cloud data to the vulnerable ROS node, resulting in buffer overflow when memcpy() function is executed. We design python-based attack scripts to simulate the hacker's actions i.e., to inject excess Point Cloud Data. We used GDB debugger to monitor memory in real-time and detect that the buffer overflow has successfully overwritten the return address leading to stack smashing (memory corruption) and system crash. The details of the simulation environment, attack script, and vulnerable C++ code, and stack results are provided in Section 3.4.1.

3.2 NFV based Virtualized Security Framework

This section discusses the requirements that drive the design of a virtualized security framework. This section also presents a high-level view of the proposed framework along with a description of the framework components.

3.2.1 Requirements

The architectural requirements of the proposed framework are described as follows:

1) Adaptability: The framework must support the implementation of code variants of CAV software as VNFs.
2) Scalability: The framework must have the ability to scale to incorporate additional services and additional code variants for the existing services.
3) Elasticity: The framework must have the ability to scale the resources as needed, ensuring efficient resource utilization.
4) Optimized deployment of VNFs: The framework must support the selection of optimal VNFs to provide maximum protection and minimize intrusion.
5) Availability of ready-to-use VNFs: The framework must provide the ability to automatically generate VNFs when needed.
6) Resiliency: Since the security environment of CAVs is evolving, the framework must provide the ability to monitor new and dynamic threats to allow the generation of resilient VNFs.
7) Fast provision of new CAV services: Our framework must support the easy integration of VNFs to offer new CAV services.
8) Distributed Network Function Virtualization Infrastructure (NFVI): Distributed NFVI is needed to support the execution of VNFs on vehicles as well as at multiple edge locations. NFVI at these locations must cooperate and communicate to ensure the VNFs representing the CAV services are provisioned as needed.
9) Dynamic VNF migration: The framework must support the dynamic migration of VNFs from one NFVI to another NFVI. This is because not all NFVIs execute the same set of VNFs. If a VNF malfunctions or becomes unavailable at an NFVI, it can be migrated from another NFVI to offer the required service.

3.2.2 High-Level View

We designed the proposed virtualized security framework considering the above requirements. Our framework is designed using the NFV paradigm. The NFV-specific components of the framework are in line with the NFV architecture standardized by ETSI (ETSI, 2021). Figure 3 shows the high-level view of

the proposed framework. The framework spans three domains: cloud, edge, and the local CAV domain. Each domain contains an NFVI (NFV Infrastructure), NFV MANO (NFV Management and Orchestration), and VNF repository. The local CAV domain includes three additional components: a code diversification controller, a code generator, and a threat assessment module. All components are described as follows:

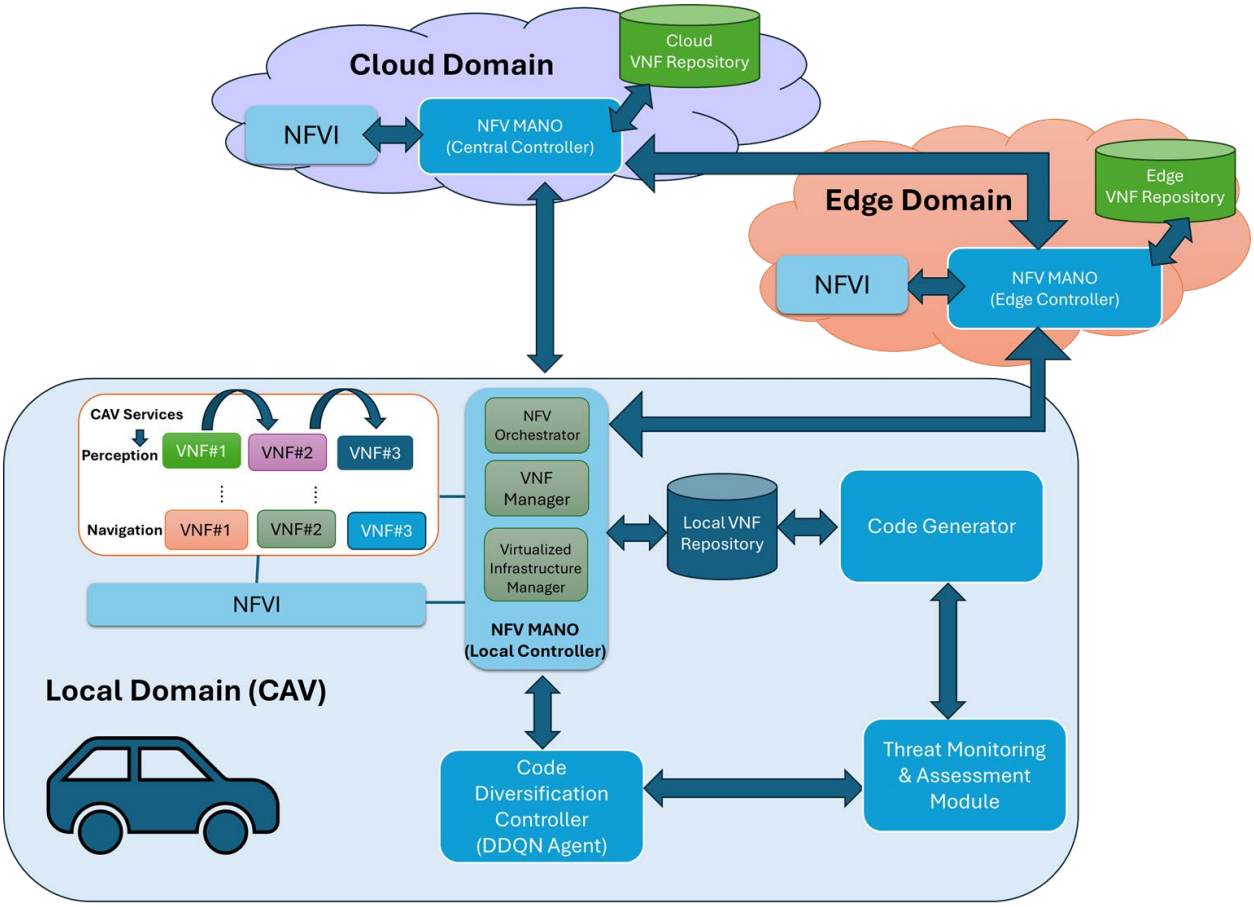

Figure 3: High-Level of the Virtualized Security Framework

1) NFV MANO: NFV MANO consists of three blocks: a) NFV Orchestrator is responsible for orchestration and management of NFVI and software resources, b) VNF Manager responsible for managing the life cycle of VNFs and performs tasks including instantiation, update, scaling, migration, and termination, 3) Virtualized Infrastructure Manager responsible for controlling and managing the computing, storage and network resources of an NFVI. The NFV-MANO in the cloud domain serves as the central controller and manages the NFVI of the local as well as edge domains. to dynamically migrate VNFs from one domain to another as needed.
2) NFVI: It provides the hardware and software resources needed to deploy, manage, and execute VNFs. In our framework, all three domains provide NFVI, thereby providing the ability to host VNFs.
3) VNF Repository: It provides a collection of VNFs that implement code variants for various CAV modules including perception, navigation, and diagnostics. VNF repositories are available in all three domains. It allows faster dissemination of VNFs designed to withstand new and dynamic threats.

4) Threat Monitoring & Assessment Module: It is responsible for monitoring unauthorized activities, including probing. This module will then perform an assessment of the threats and predict the likelihood of exploiting vulnerabilities in CAV software.
5) Code generator module: It is responsible for generating multiple versions in the same language as well as in a different language based on the output of the threat monitoring and assessment module.
6) Code Diversification Controller: This controller implements a code diversification mechanism to improve resiliency of CAV software. The code diversification mechanism allows dynamic reconfiguration of the CAV software modules by switching between functionally equivalent code variants, implemented as VNFs. The optimal VNFs are the ones that maximize resiliency and uptime and minimize the cost of resources used by VNFs. We design the controller using the DDQN agent that learns to optimize the selection of VNFs based on the reward it receives from the CAV's software environment. The reward function is designed by considering several key parameters including the ability of the agent to maintain the service continuity, preventing intrusions, and the cost incurred by the execution of the VNFs. The details of the DDQN agent are described in section 3.3.

3.3 Optimal Code Diversification Using Reinforcement Learning

We developed RL agents (Q-learning and DDQN) to select the optimal VNF to be executed next during the switching, a problem referred to as optimal VNF Switching. Our DDQN agent is trained to learn an optimal switching policy that enhances the system's security posture. The defense environment is modeled as a Markov Decision Process (MDP), where the RL agent interacts with a custom simulated execution environment, **CAVExecutionEnv-v3**. Through ongoing interaction with its environment, the agent adapts the execution sequence of code variants in real time, optimizing defense performance by minimizing security breaches, improving resilience against intrusions, and maintaining service continuity in the face of cyber threats. The reward function is designed with a multi-objective focus. It penalizes configurations associated with high security risks (e.g., based on Common Vulnerability Scoring System [CVSS] metrics), observed intrusion events, and unnecessary switching frequency, while rewarding actions that yield high security performance with minimal system overhead.
The following sections detail the formal MDP formulation, including the state, action, and reward design, along with the training strategy and experimental framework for both DDQN and baseline Q-learning agents.

3.3.1 Environment and MDP Formulation

The optimal switching of VNFs problem is formulated as a Markov Decision Process (MDP), which provides a rigorous mathematical framework for sequential decision-making in dynamic environments. To simulate real-world adversarial conditions, a custom RL environment, **CAVExecutionEnv-v3,** was developed using Python 3.9 and executed on a Google Colab CPU runtime. This environment models the operational behavior of CAV software under diverse cyberattack scenarios and enables interaction with the DDQN agent. The MDP is defined by the following components:

3.3.2 State Representation Design

Each state represents an 8-dimentional continuous vector ($S_t \in \mathbb{R}^8$), providing a profile of the CAVs security posture and system behavior at time step *t.* The features observed by the agent, precisely as generated in **CAVExecutionEnv-v3**, are:

S_1: Normalized Time step. $\frac{Current\ step}{Maximum\ steps}$

S_2 : Normalized Switch count, total number of software configuration switches normalized by the maximum allowed $\frac{Switches}{Maximum\ switches}$

S_3 Basic Intrusion Rate, The rate of successful basic intrusions. $\frac{Basic\ Intrusion\ counter}{Max(1, current\ step)}$

S_4 : Attack attempt Rate, Frequency of adversarial attack attempts over time. $\frac{Attack\ attempt}{Max(1, current\ step)}$

S_5 Complex intrusion Rate, the rate of complex intrusions. $\frac{Complex\ Intrusion\ counter}{Max(1, current\ step)}$

S_6 Defense success Rate, Proportion of attacks that were successfully defended. $\frac{Defended\ attacks}{Max(1, attack\ attempts)}$

S_7 Normalized Resource Cost, An aggregated and normalized measure of the CPU, RAM, and Storage consumption of the currently active VNFs $\frac{Total\ resource\ cost}{100}$

S_8 Entropy Noise, A random float between 0 and 1, introducing stochasticity into the observation space.

3.3.3 Action Space Modeling

At each time step t, the agent would select an action A_t from a MultiDiscrete action space, meaning the agent simultaneously picks VNF for each critical module: Perception, Navigation, and Diagnostics, forming an action vector.

$$A_t = [a_1, a_2, a_3]$$

For example, an action corresponds to the selection of a VNF:C++-v1 for perception, VNF: Java-v2 for Navigation, and VNF: Python-v1 for Diagnostics. This combined module-language reconfiguration directly impacts the security posture of the system (via CVSS-based vulnerability exposure) and operational overhead (via resource consumption profiles). To integrate this Multidiscrete action space with the DDQN agent, which expects a single discrete output, the action vector is linearized into a single discrete space before being passed to the agent and then re-mapped to the Multidiscrete format within the environment for execution.

3.3.4 Reward Function Design

The reward function R_t is carefully designed to guide the RL agent into making decisions that enhance both security resilience and operational efficiency. At each time step T, the total reward is computed as a composite of multiple incentives and penalties and is expressed as:

$$R_{t=} = R_{base} + R_{proactive} + R_{security\ bonus} + R_{switch\ security\ bonus} - p_{risk} - p_{resource} - p_{switch} - P_{attack} \quad (1)$$

Where:

- R_{base}: A constant positive reward to the agent for maintaining operational continuity.

- $R_{proactive}$: Awarded if an attack was not successful, scaled by the inverse of the current average CVSS risk, encouraging the agent to maintain low-risk configurations.
- $R_{securityBonus}$: A bonus for successfully mitigating known or complex intrusions.
- $R_{switchSecurityBonus}$: Bonus associated with secure language or module switching under attack conditions.
- P_{risk}: Penalty proportional to the current module's risk profile, such as CVSS-based vulnerability scores.
- $P_{resource}$: Penalty reflecting the computational cost of a VNF, encouraging resource-efficient configurations.
- P_{switch}: A minor penalty applied for each VNF switch performed at the current step, discouraging excessive or unnecessary reconfigurations.
- P_{attack} Penalty imposed upon successful intrusion or security breach

The total reward computed at each step is normalized using a fixed scaling factor to maintain training stability and avoid Q-value divergence. The scaled reward is expressed as:

$$R'_t = \lambda \times R_t, \quad (2)$$

where $\lambda = 0.2$

3.3.5 Q-Learning Baseline Agent

We proposed a Q-Learning-based adaptive defense mechanism to optimally switch between code variants (Java, Python, and C++) to improve system resiliency against cyber-attacks (Benibo et al., 2025). The QL agent chooses the optimal code variant using a reward function that penalizes unnecessary transitioning and encourages security and stability.

We extended the Markov Decision Process (MDP) used for Q-learning to capture multiple aspects of the observation space including dynamic intrusion rates, resource utilization by VNFs, and defense metrics, while adapting the action space to a multi-discrete reconfiguration across critical modules in CAV. The reward function was then restructured to reflect multi-objective goals including proactive adaptation, security resilience, and operational efficiency. To provide a comprehensive benchmark, this tabular Q-learning agent was trained against our **CAVExecutionEnv-v3.** Q-learning is a model free reinforcement RL algorithm that learns an action-value function representing expected outcome for taking an action in state.

The foundation of Q-Learning is its update rule, which iteratively refines the estimated Q-values (state-action values). For a given state-action pair (s, a), the Q-values are updated based on the reward (r) received and the maximum Q-value of the next state (s'):

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \quad (3)$$

Where:

- α is the learning rate
- γ is the discount factor

- r is the intermediate reward
- s is the current state
- a is the action taken at the current state
- s′ is the next state, and
- a′ is the action taken at the next state.

The Q-Learning agent discretizes the continuous 8-Dimentional state space for the **CAVExecutionEnv-v3** into a manageable number of bins for each feature, given room for the environment to be represented in a finite state-action pair space. Using the Bellman equation (3), the agent updates a Q-table (a look-up table storing $Q(s,a)$ values). This baseline method is instrumental in understanding the dynamics of the **CAVExecutionEnv-v3** and serves as an important point of comparison for more advanced deep reinforcement learning. The setup was designed to align with the environment structure, including its Multidiscrete action space, (through linear mapping to a single discrete action for tabular representation), CVSS-based vulnerability modeling, and the multi-component reward function. The normalized observation space provided by the **CAVExecutionEnv-v3** was binned to fit the tabular approach.

3.3.6 DDQN Agent Design

The DDQN agent was implemented to address the limitations faced by tabular Q-Learning, precisely its inability to scale high-dimensional, continuous state spaces and its sensitivity to overestimation bias in dynamic environments. Unlike the tabular Q-Learning that stores Q (s, a) values in a table, DDQN uses deep neural network function approximators for Q-values estimation, that effectively overcomes the scalability issues faced by the tabular methods.

The main innovation of the DDQN over the DQN is its method for calculating the target Q-value that helps in the mitigation of the overestimation bias faced by the standard Q-Learning (Halat et al., 2024).. For a traditional DQN, the same network is used for both action selection and value evaluation, which can lead to an overestimation of Q-value. DDQN mitigates this by decoupling the action selection from the evaluation process. Using one network (Online) is used for action selection, while another network (target) is used for evaluating the value of that action.

The target Q-value in DDQN is calculated as:

$$Y_t^{\text{DDQN}} = r + \gamma Q_{\text{target}}\left(s', \arg\max_a Q_{\text{policy}}(s', a)\right) \quad (4)$$

Where:

- y_t^{DDQN} is the target Q-value for the DDQN update.
- r is the immediate reward.
- γ is the discount factor
- Q_{target} The Q-value is estimated by the target network.
- Q_{policy} The Q-value is estimated by the online Online Q-Network.
- $\arg\max_a Q_{\text{policy}}(s', a)$ is the action selected by the Online Q-Network for the next state s'

The purpose of using a separate network is to ensure that the action is chosen based on the updated policy, while its value is estimated more conservatively by a table, older version of the network (target network), therefore reducing the overestimation issue and improving learning stability.

The DDQN architecture comprises of the following key components:

- **Two Neural Networks:** The core of the DDQN consists of an online Q-Network network and a target network. Both are multilayered perceptron (Q-Network) with an input size aligning with the environment's 8-dimensional state and an output size equal to the size of the linearized MultiDiscrete action space. Each network has two hidden layers with 256 neurons each, using the Rectified Linear Unit (ReLU) activation function (F.relu) for non-linearity. Weights are initialized using Xavier uniform initialization, and biases are set to zero, enhancing stable training.
- **Experience Replay Buffer:** A deque of maxlen=10000 serves as the experience replay buffer. This mechanism stores past transitions and randomly samples batches during learning. This breaks the temporal correlations inherent in sequential experiences, therefore improving sample efficiency and stabilizing the learning process.
- **Target Network Updates:** The target network (target_net) is a periodically updated copy of the Online Q-Network (policy_net). Instead of continuous updates, the target network's weights are hard copied from the Online Q-Network every 100 steps (update_target_every = 100). This delayed update mechanism stabilizes the target Q-value's estimates, thereby reducing oscillations and improving convergence.
- **Optimization and Loss Function:** The DDQN agent uses Adam optimizer (optim.Adam), with a learning rate of 0.001 to minimize the loss function. The loss function deployed is the Mean Squared Error (MSE)(F.mse_loss), which measures the squared difference between the predicted Q-value from the Online Q-Network, Q_policy (s, a), and the target Q-value $\left(Y_t^{\mathrm{DDQN}}\right)$. To prevent gradients exploding, gradient clipping is applied via torch.nn.utils.clip_grad_norm_ with a maximum norm of 1.0.

The DDQN agent outputs action-value estimates for all possible module-VNF configurations, allowing it to learn optimal switching strategies based on observed attack patterns and the nuanced reward trends provided by the environment. Figure 4 illustrates the DDQN architecture, showing the interaction between the environment, the online Q-network, the target Q-network, and the replay memory. It highlights the separated action selection and value estimation process, and the periodic copying of parameters from the online network to the target network, which collectively mitigate the bias of overestimation.

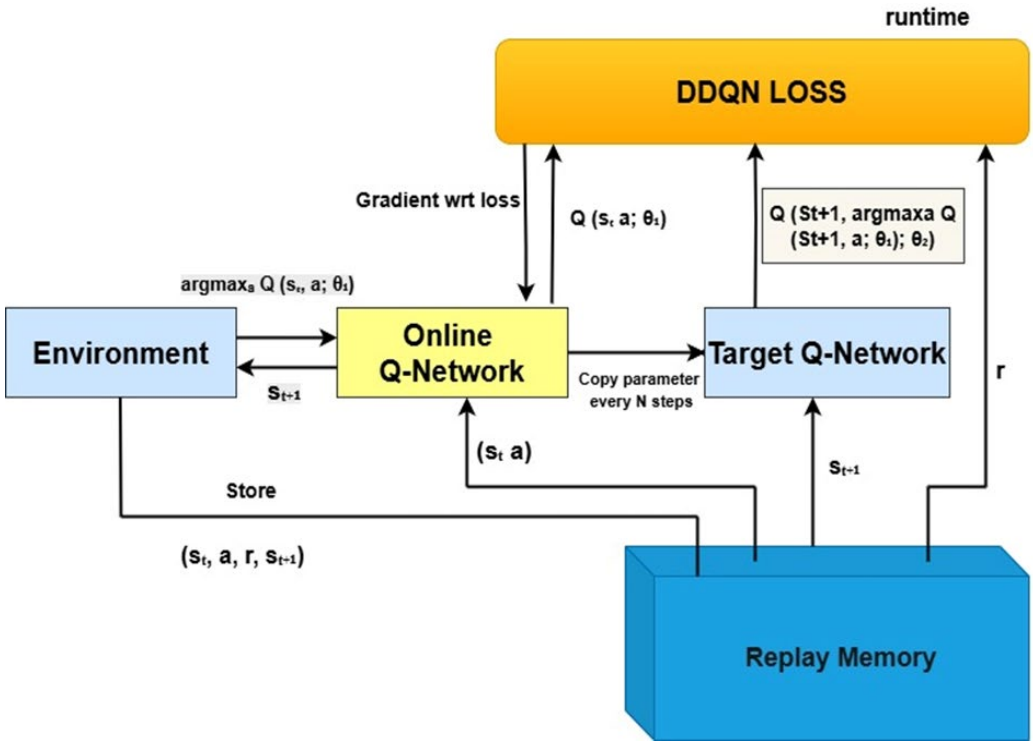

Figure 4: Schematic illustrating the Double Deep Q-Network (DDQN) architecture and loss calculation process.

3.4 Experimental Methods

3.4.1 Exploitation Method for Buffer Overflow

This section discusses experimental methods for exploiting buffer overflow vulnerability. We conducted static analysis to identify vulnerabilities in C++ modules in the Autoware environment. We then designed a vulnerable C++ program for combining point cloud data and executed the program under both normal and attack scenarios.

3.4.1.1 Threat Model

An attacking scenario where the attacker is capable of injecting malformed Lidar data via a topic using ROS2 DDS is considered. The attacker does not require root access to the target system but has sufficient permission to publish sensor messages. The goal is to trigger a buffer overflow attack in a vulnerable ROS node that is responsible for concatenating point cloud data. We considered the memcpy () function to demonstrate the buffer overflow attack.

3.4.1.2 Simulation Environment: AWSIM and Autoware Integration

To show a realistic autonomous driving environment, we integrate AWSIM (*AWSIM Document*, n.d.) , an open-source traffic simulator, with Autoware (*The Autoware Foundation · GitHub*, n.d.), a widely used ROS2-based software stack for autonomous vehicles shown above. AWSIM simulates sensor data, traffic, road infrastructure, and dynamic entities (e.g., NPC vehicles, pedestrians, traffic lights), while Autoware

processes synthetic sensor data through its core modules (sensing, localization, perception, planning and control).The ego vehicle in AWSIM is equipped with numerous sensors like Lidar and camera, whose outputs are fed into Autoware's perception pipeline. The communication between AWSIM and Autoware is achieved through using a local ROS2 network.

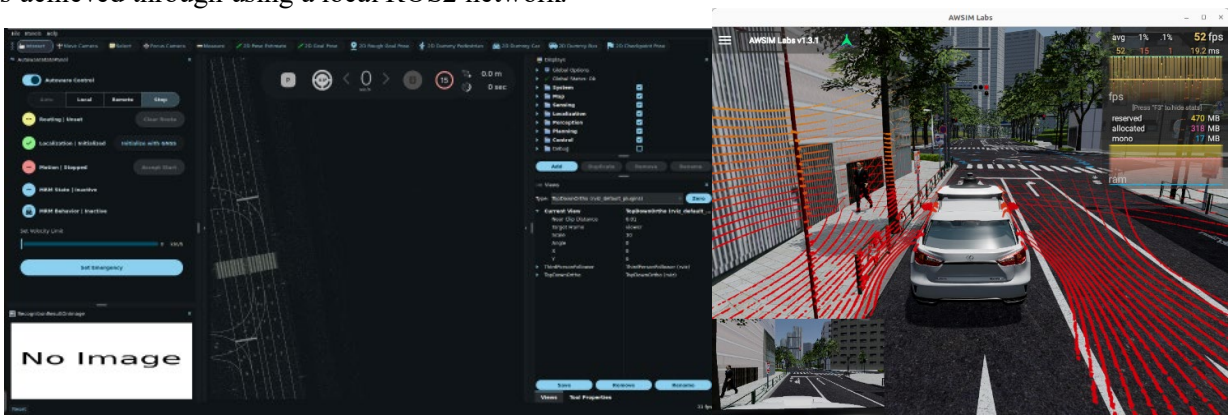

Figure 5: AWSIM AND Autoware CAV simulation environments

3.4.1.3 Static Vulnerability Analysis Using Flaw finder

Flaw finder (*Flawfinder Home Page*, n.d.) is used to scan the Autoware source code for unsafe C/C++ function calls, particularly focusing on memory manipulation operations such as memcpy, strcpy, and strcat. The analysis flags vulnerabilities and assigns severity levels (0–5) based on the Common Weakness Enumeration (CWE) taxonomy (MartinBob, 2019).

Table 1: Vulnerability risk levels and densities identified by flawfinder in the entire Autoware software stack

Risk Level	Number of Risks	Density (Hits/KSLOC)
0 (Low)	2393	3.095
1 (Low-Moderate)	196	1.881
2 (Moderate)	3452	1.782
3 (High)	28	0.031
4 (Critical)	32	0.017
5 (Very Critical)	1	0.001

The above table summarizes the risk levels and vulnerability densities detected across the Autoware stack. Notably, CWE-120 (“Buffer Copy without Checking Size of Input”) was found to be the most critical and relevant to our study.

One of the files that contained the CWE-120 flag, the occupancy_grid_map_outlier_filter_node.cpp, was further isolated and analyzed. This node was flagged with multiple moderate-risk vulnerabilities like the unbounded use of memcpy() in Lidar data concatenation routine. A summary of the analysis is shown below.

Table 2: Analysis Metrics Summary of occupancy grid map outlier node

Metric	Value
Total Vulnerabilities Detected	20
Analysis Time	0.06 seconds
Lines of Code (SLOC)	464
Processing Speed	9,869 lines/sec
Risk Level	2 (Medium)
Hits/KSLOC	43.1034

3.4.1.4 Exploit Design: Lidar Buffer Overflow Attack

We developed a custom C++ node that combines point cloud data from multiple lidar sensors and produces the final point cloud. The C++ point cloud concatenator node (as shown in Figure 6) serves as the attack surface designed to combine lidar data streams from three independent publishers (\sensing\lidar\topic1, \topic2, \topic3) into a single point cloud. Its implementation introduces memory corruption vulnerability through a fixed-size buffer of 256 bytes and the absence of bound checking during data concatenation.
Each Python node (shown in Figure 7) publishes point cloud messages with geometric coordinates (horizontal, vertical and depth-aligned points), also including other parameters such as width, point step, and data fields. When the C++ node subscribes to these streams, it copies each point cloud's raw byte data into the undersized buffer using the memcpy() function without validating the cumulative size of incoming data. This flaw allows the Python scripts to overflow the buffer by generating malformed lidar point clouds. By executing attacks across all three topics, the total size (480 bytes) exceeds the buffer's 256-byte capacity, triggering a stack overflow that corrupts adjacent memory regions. To exploit the buffer overflow vulnerability, the custom C++ was launched together with three Python nodes (attack scripts).

Algorithm Point Cloud Concatenator C++

```
procedure PointCloudConcatenator
    Initialize three subscriptions for topics:
        "/sensing/lidar/topic1"
        "/sensing/lidar/topic2"
        "/sensing/lidar/topic3"
    Initialize publisher for topic "/sensing/lidar/concatenated/pointcloud"
end procedure
procedure Try_Combine
    if buffer[0] is NULL OR buffer[1] is NULL OR buffer[2] is NULL then
        return // Wait until all buffers have data
    end if
    Define UNSAFE_BUFFER_SIZE = 256
    Initialize unsafe_buffer as array of size UNSAFE_BUFFER_SIZE
    Set buffer.offset = 0
    Create combined_cloud as PointCloud2 object
    Set combined_cloud.header = buffer[0].header
    Set combined_cloud.header.stamp = current time
    Set combined_cloud.height = 1
    Set combined_cloud.fields = buffer[0].fields
    Set combined_cloud.is_bigendian = buffer[0].is_bigendian
    Set combined_cloud.point_step = buffer[0].point_step
    Store cloud in buffer
    Copy cloud.data into unsafe_buffer starting at buffer.offset
    Increase buffer.offset by size of cloud.data
    Assign data from unsafe_buffer to combined_cloud.data
    Set combined_cloud.width = buffer.offset / combined_cloud.point_step
    Set combined_cloud.row_step = buffer.offset
    Publish combined_cloud on "/sensing/lidar/concatenated/pointcloud"
    Reset buffer = {NULL, NULL, NULL}
    Reset buffer.offset = 0
end procedure
procedure Main
    Initialize ROS2
    Start PointCloudConcatenator
    Spin until shutdown
    Shutdown ROS2
end procedure
```

Figure 6: Point Cloud Concatenator node.

Algorithm Point Cloud Publishers (1,2,3)

```
procedure POINTCLOUDPUBLISHER(topic, number)
    Initialize ROS2 node named pc_publisher.topic_number
    Subscribe to topic "/sensing/lidar/topic_number"
    Initialize fields:
      (1) width = 10
      (2) height = 1
      (3) point_step = 16
      (4) is_bigendian = False
      (5) row_step = width * point_step
      (6) fields = {x, y, z, intensity}
end procedure
procedure PUBLISH_POINTS
    Create PointCloud2 message msg
    Set msg.header.stamp = current time
    Set msg.header.frame_id = "base_link"
    Initialize empty list data
    for i = 0 to 9 do
       append (x, y, z, 150, 2, 2) to data
    end for
    msg.data = data
    Publish msg using publisher
    // Loop topicNumber × 10 horizontal/vertical/depth points
end procedure
procedure MAIN
    Start POINTCLOUDPUBLISHER with appropriate topic and number
    Spin until shutdown
    Shutdown ROS2
end procedure
```

Figure 7: Python Point Cloud Publisher.

3.4.1.5 Runtime Verification with GDB

The vulnerable C++ node is run with GNU Debugger (GDB) to inspect the stack frames before and after launching the attack. GDB is a debugging tool that enables step-by-step program execution, memory and variables inspection, and program examination behavior at the machine level. The GDB workflow is shown in Figure 8.

```
Running Point Cloud Concatenator C++ with GDB

initialize_gdb() {
  target ← "~/ros2_ws/install/my_pkg/lib/my_pkg/concatenator_node"
  command ← "gdb ${target}"}

set_breakpoints() {
  command ← [
    "break 1,
    "break 2  ]}
start_execution() {
  command ← "run"}
inspect_stack_before_memcpy() {
  command ← [
    "info frame",
    "info registers",
    "p &unsafe_buffer",
    "p $rbp - $rsp",
    "x/**x $rsp"]}
run_attack_scripts() {
  execute ← [
    "pc_publisher_1.py",
    "pc_publisher_2.py",
    "pc_publisher_3.py"]}
continue_execution_past_memcpy() {
  command ← "continue"}
catch_segmentation_fault() {
  signal ← "*** stack smashing detected *** → SIGABRT"}
  command ← [
    "info frame",
    "info registers",
    "p $rbp - $rsp",
    "x/**x $rsp",
     "bt"]}
final_analysis() {
  compare ← "saved rip before vs after"
  observe ← "float encoded values overflowed stack"}
```

Figure 8: GDB workflow

3.4.2 NFV Based Virtualized Security Framework Experimental Setup

This section details the prototype architecture: NFVI setup & OpenStack Deployment, and performance measures deployed in our experimental testbed. The purpose of this setup is to build a prototype of the proposed virtualized framework and demonstrate the benefit of code diversification under simulated memory corruption attacks.

3.4.2.1 Prototype architecture: NFVI Setup and OpenStack Deployment

We designed and deployed a prototype of our virtualized security framework leveraging OpenStack platform. The prototype implements the components: NFVI, NFV MANO, VNFs, and a simplified code diversification controller. Figure 9 shows our prototype architecture. OpenStack's 5.2.0 MicroStack was installed as a single node to create an NFVI with both the controller and compute node.

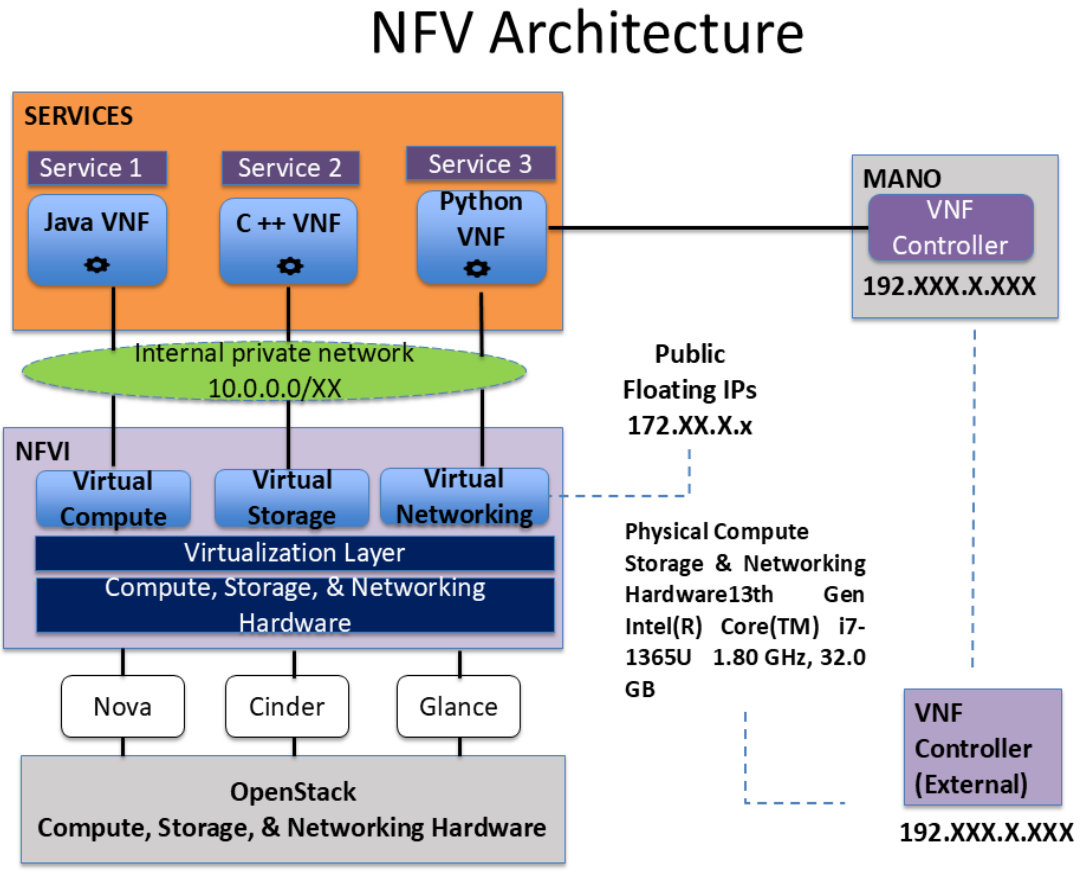

Figure 9: Prototype architecture.

The NFVI is hosted on a dedicated physical machine running Ubuntu 22.04. The NFVI operates over Wi-Fi using static IP configurations and can perform VNF orchestration. The core OpenStack services Keystone (identity), Glance (image), Nova (compute), Neutron (network), Horizon (dashboard), and Cinder (block storage) were configured with internal and external (floating) IP networking enabled through Wi-Fi-based br-ex integration.

Three VNFs were instantiated as independent Ubuntu-based virtual machines (VMs) within the single-node MicroStack installation. These VNFs implement the functionality of point cloud concatenator node in three programming languages as follows:

- Java VNF: Executes a Java implementation using JeroMQ, a pure Java ZeroMQ library.
- C++ VNF – Runs a C++ point cloud concatenator compiled as a standalone binary.
- Python VNF: Runs a Python-based concatenator.

Each VNF communicates using ZeroMQ over TCP sockets. We used python-based scripts to publish 3D point cloud fragments mimicking LIDAR-like data (horizontal, vertical, and depth variations). For the sake of simplicity, the publishers share the same Ubuntu VM that runs the Python VNF.

3.4.2.2 Performance Metrics

We used the following performance metrics to assess the resource usage by the controller and some of the VNFs (C++ and Java) of our framework. The following metrics were assessed.

- Attack Success Rate (%): This is the percentage of attacks that were successful. This metric was measured with respect to various switching intervals.
- CPU usage: This is the time the CPU spent executing user space % usr and kernel space % sys code for VNFs. Also, total CPU load per process is represented by % CPU. CPU Usage was collected every 5 seconds inside the VMs used for C++ VNF and Java VNF.
- Memory Usage (RSS): This is the resident set size (RSS) of each VNF process in MiB, measuring the actual physical memory used. This was logged per process per core.

3.4.3 Experimental Setup for DDQN Agent

Our DDQN agent is trained using the CAVExecutionEnv-v3 described in section 3.3.3. We simulated a dynamic CAV environment where the DDQN agent's performance was evaluated according to its response to Common Vulnerability Scoring System (CVSS)-based vulnerability scores. The primary aim was to assess how effectively the agent can learn optimal software switching strategies to balance aggressive attack avoidance while minimizing resource usage, leveraging CVSS risk as a critical component in its decision-making process. The hyperparameters used during training and validation, vulnerability metrics, and the performance metrics are discussed in the following sections.

3.4.3.1 DDQN Agent's Hyperparameters

Table 1 shows the hyperparameters for the neural networks of the DDQN agent. Table 1 also shows the hyperparameters including soft update rate and replay buffer size that are specific to the DDQN algorithm.

Table 3: DDQN Agent's Hyper-parameter.

Parameter	Value / Description
Neural Network Architecture	2 hidden layers, 256 neurons each
Activation Function	ReLU
Optimizer	Adam
Learning Rate	0.001
Loss Function	Mean Squared Error (MSE)
Experience Replay Buffer Size	10,000
Batch Size	64
Discount Factor (γ)	$\gamma = 0.99$
Initial Epsilon (ϵ_0)	$\epsilon_0 = 1.0$
Minimum Epsilon (ϵ_{min})	$\epsilon_{min} = 0.02$
Epsilon Decay Rate (ϵ_decay)	Decay = 0.995 per episode
Target Network Soft Update Rate (τ)	$\tau = 0.005$
Target Network Sync Strategy	Soft update after every learning step
Action Selection Policy	ε-greedy exploration strategy
Exploration Device Support	CUDA / CPU automatically detected

3.4.3.2 CVSS v3.1 Vulnerability Metrics

To fully define the environment's vulnerability model, the specific CVSS v3.1 metrics for the vulnerabilities considered in CAVExecutionEnv-v3 are detailed in the table below. These values directly influence the risk penalties and proactive defense bonuses within the reward function, as discussed in Section 3.3.5

Table 4: CVSS v3.1 Vulnerability Metrics and Characteristics.

CWE-ID	CWE Name	CVSS Severity	Typical Impact	Impact Score	Common Languages
CWE-122	Heap-based Buffer Overflow	8.8 (High)	Memory corruption	6.0	C/C++
CWE-95	Improper Neutralization of Directives in Dynamically Evaluated Code	9.8 (High)	Arbitrary code execution	5.9	Java, Python
CWE-502	Deserialization of Untrusted Data	7.8 (Critical)	RCE, Data tampering	5.9	Java, Python
CWE-611	Improper Restriction of XML External Entity References	5.5 (Medium)	Denial of Service (DoS)	5.3	Java
CWE-94	Improper Control of Generation of Code	7.2 (High)	Arbitrary code execution	5.9	Python, Java
CWE-787	Out-of-bounds Write	7.1 (Critical)	Memory corruption	6.0	C++

3.4.3.3 Performance Metrics

We used the following performance metrics to evaluate the performance of our DDQN agent.

a) Quality of Service (i.e., Uptime) (%): This measures the percentage of time the system remained uncompromised as a result of cyberattacks or defense mechanisms. It indicates the system's availability and reliability.

b) Mean Time Before Intrusion (MTBI): This metric represents the average number of steps between consecutive successful intrusions. A higher MTBI value indicates better system resilience and longer periods of uninterrupted secure operation. This metric is computed as Total steps/Total Intrusions.

c) Reward: This metric represents the reward value accumulated by the RL agent. We measure both episodic and cumulative reward attained by the agent.

d) Security Success Rate (%): This metric represents the percentage of attack attempts that were successfully prevented by the adaptive defense mechanism. This is computed as (Defended Attacks / Total Attack Attempts) × 100

CHAPTER 4

Results

4.1 Buffer Overflow results

The lidar buffer overflow exploitation involved running the vulnerable C++ script using GDB and monitoring the stack state before and after the attack. Our stack observations are explained below.

4.1.1 Normal operation state

During normal execution of the try_combine() function (as shown in Figure 6) in the pointcloud_concatenator node, the runtime stack adheres to standard x86_64 calling conventions, with the stack frame established between the stack pointer (rsp = 0x7fffffffc690) and base pointer (rbp = 0x7fffffffc850). Within this frame, a fixed-size local buffer is defined as uint8_t unsafe_buffer[256], with a corresponding variable buffer_offset initialized to zero. This array is automatically placed in the stack segment each time the function is called. The stack pointer (rsp) is observed at 0x7fffffffc690, while the base pointer (rbp) anchors the frame at 0x7fffffffc850, establishing a total frame size of 448 bytes. The vulnerable buffer occupies the lowest memory in this frame (approximately 0x7fffffffc690 to 0x7fffffffc790), followed closely by saved callee registers and critically the saved return address.

At address 0x7fffffffc858, GDB showed saved rip = 0x55555556f17b. This value represents the exact instruction pointer in the calling function to which execution must return upon completion of try_combine(). This address is essential for correct control flow. The saved rip is automatically pushed onto the stack during the call instruction and later restored during the ret instruction.

To evaluate the behavior around this sensitive area, two breakpoints were strategically set using GDB; one immediately before the call to memcpy() (line 55) and another after its execution (line 70). At the first breakpoint, the stack inspection (x/448x $rsp) revealed that the saved rip and adjacent registers (rbp, rbx) were intact, and that the buffer boundaries did not intrude into the control data region.

```
Thread 1 "concatenator_no" hit Breakpoint 1, PointCloudConcatenator::try_combine (
    ↪ this=0x5555556662c0) at /home/user/ros2_ws/src/my_pkg/src/
    ↪ pointcloud_concatenator.cpp:55
55          std::cout << "Before memcpy"<< std::endl;
(gdb) info frame
Stack level 0, frame at 0x7fffffffc860:
 rip = 0x55555556fd82 in PointCloudConcatenator::try_combine (/home/user/ros2_ws/
    ↪ src/my_pkg/src/pointcloud_concatenator.cpp:55); saved rip = 0x55555556f17b
 called by frame at 0x7fffffffc880
 source language c++.
 Arglist at 0x7fffffffc850, args: this=0x5555556662c0
 Locals at 0x7fffffffc850, Previous frame's sp is 0x7fffffffc860
 Saved registers:
  rbx at 0x7fffffffc848, rbp at 0x7fffffffc850, rip at 0x7fffffffc858
(gdb) p &unsafe_buffer
$1 = (uint8_t (*)[256]) 0x7fffffffc730
(gdb) info registers
rax 0x5555556dd4d0 93824993842384 rbx 0x8 8 rcx 0x5555556dd550 93824993842512
rdx 0x0 0 rsi 0x0 0 rdi 0x7fffffffc6d0 140737488340688
rbp 0x7fffffffc850 rsp 0x7fffffffc690 r8 0x0 r9 0x5555556dd550
r10 0x0 r11 0x7ffff781ace0 r12 0x5555556d76a8 r13 0x7fffffffd030
r14 0x7fffffffcc30 r15 0x55555564b288 rip 0x55555556fd82
eflags 0x206 [ PF IF ] cs 0x33 ss 0x2b ds 0x0 es 0x0 fs 0x0 gs 0x0
(gdb) p $rbp - $rsp
$2 = 448
(gdb) x/448x $rsp
0x7fffffffc690: 0x00000001 0x00000000 0x556662c0 0x00005555
0x7fffffffc6a0: 0x00000000 0x00000000 0xf7fbb4c0 0x00007fff
0x7fffffffc6b0: 0x00000100 0x00000000 0xf7fbb830 0x00007fff
0x7fffffffc6c0: 0x00000001 0x00000000 0x00000000 0x00007fff
0x7fffffffc6d0: 0x556dd4d0 0x00005555 0x556dd4c0 0x00005555
0x7fffffffc6e0: 0xf7fb49e0 0x00007fff 0xf3a04e2b 0x18487ae5
0x7fffffffc6f0: 0x00000001 0x00007fff 0xffffc758 0x00007fff
0x7fffffffc700: 0xf7dafb4c 0x00007fff 0xffffd480 0x00007fff
0x7fffffffc710: 0x556d7540 0x00005555 0xffffd628 0x00007fff
0x7fffffffc720: 0x5564b288 0x00005555 0xf7d4e17a 0x00007fff
0x7fffffffc730: 0x00000000 0x00000000 0x556ce140 0x00005555
0x7fffffffc740: 0x556d7540 0x00005555 0xf7d9930b 0x00007fff
0x7fffffffc750: 0xffffffff 0x7fffffff 0xf7d55e1e 0x00007fff
0x7fffffffc760: 0x00000001 0x00000000 0xf7f46d50 0x00007fff
0x7fffffffc770: 0xffffd620 0x00007fff 0xf7da8848 0x00007fff
0x7fffffffc780: 0x00000000 0x00000000 0xf7da867e 0x00007fff
0x7fffffffc790: 0x556c9470 0x00005555 0xffffffff 0xffffffff
0x7fffffffc7a0: 0xffffcbe0 0x00007fff 0x556dd300 0x00005555
0x7fffffffc7b0: 0xffffd030 0x00007fff 0x556ca070 0x00000001
0x7fffffffc7c0: 0xffffffff 0x00000001 0x556dd308 0x00005555
0x7fffffffc7d0: 0x0039f6aa 0x00005550 0x556dd308 0x00005555
0x7fffffffc7e0: 0xffffc810 0x00007fff 0x55572ef9 0x00005555
0x7fffffffc7f0: 0xffffc898 0x00007fff 0x55666668 0x00005555
0x7fffffffc800: 0xffffc978 0x00007fff 0x556dd300 0x00005555
0x7fffffffc810: 0xffffc830 0x00007fff 0x5556ee05 0x00005555
0x7fffffffc820: 0xffffc890 0x00007fff 0x55666660 0x00005555
0x7fffffffc830: 0xffffc850 0x00007fff 0xfae82b00 0xf4f52495
0x7fffffffc840: 0xffffc890 0x00007fff 0x556d76a8 0x00005555
0x7fffffffc850: 0xffffc870 0x00007fff 0x5556f17b 0x00005555
```

Figure 10: Stack state at Breakpoint 1 before memcpy() execution in try_combine() during normal operation.

After stepping past the memcpy() instruction, the second breakpoint confirmed that, under normal input conditions, the value at 0x7fffffffc858 remained unchanged, ensuring correct saved rip = 0x55555556f17b.

```
Continuing. Before memcpy After memcpy
Thread 1 "concatenator_no" hit Breakpoint 2, PointCloudConcatenator::try_combine (
    ↪ this=0x5555556662c0) at /home/user/ros2_ws/src/my_pkg/src/
    ↪ pointcloud_concatenator.cpp:70
70          combined_cloud->data.assign(unsafe_buffer, unsafe_buffer +
    ↪ buffer_offset);
(gdb) info frame
Stack level 0, frame at 0x7fffffffc860:
 rip = 0x55555556fec4 in PointCloudConcatenator::try_combine (/home/user/ros2_ws/
    ↪ src/my_pkg/src/pointcloud_concatenator.cpp:70); saved rip = 0x55555556f17b
 called by frame at 0x7fffffffc880
 source language c++.
 Arglist at 0x7fffffffc850, args: this=0x5555556662c0
 Locals at 0x7fffffffc850, Previous frame's sp is 0x7fffffffc860
 Saved registers:
  rbx at 0x7fffffffc848, rbp at 0x7fffffffc850, rip at 0x7fffffffc858
(gdb) p &unsafe_buffer $3 = (uint8_t (*)[256]) 0x7fffffffc730
(gdb) info registers
rax   0x55555564b2c0 93824993243840    rbx   0x50            80
rcx   0x00000c00     3072              rdx   0x7ffff7c23370  140737350087536
rsi   0x0            0                 rdi   0x7ffff781ca70  140737345866352
rbp   0x7fffffffc850 0x7fffffffc850    rsp   0x7fffffffc690  0x7fffffffc690
r8    0x0            0                 r9    0x5555556dd680  93824993842816
r10   0x7ffff760d560 140737343706464   r11   0x293           659
r12   0x5555556d76a8 93824993818280    r13   0x7fffffffd030  140737488343088
r14   0x7ffffffffcc30 140737488342064  r15   0x55555564b288  93824993243784
rip   0x55555556fec4 0x55555556fec4 <PointCloudConcatenator::try_combine()+882>
eflags         0x206              [ PF IF ]
cs             0x33               51
ss             0x2b               43
ds             0x0                0
es             0x0                0
fs             0x0                0
gs             0x0                0
(gdb) p $rbp - $rsp $4= 448
(gdb) x/448x $rsp
0x7fffffffc690: 0x00000001      0x00000000      0x556662c0      0x00005555
0x7fffffffc6a0: 0x000000f0      0x00000000      0x55666670      0x00005555
0x7fffffffc6b0: 0x00000100      0x00000000      0x55666640      0x00005555
0x7fffffffc6c0: 0x55666670      0x00005555      0x55666660      0x00005555
0x7fffffffc6d0: 0x556dd4d0      0x00005555      0x556dd4c0      0x00005555
0x7fffffffc6e0: 0xf7fb49e0      0x00007fff      0xf3a04e2b      0x18487ae5
0x7fffffffc6f0: 0x00000001      0x00007fff      0xffffc758      0x00007fff
0x7fffffffc700: 0xf7dafb4c      0x00007fff      0xffffd480      0x00007fff
0x7fffffffc710: 0x556d7540      0x00005555      0xffffd628      0x00007fff
0x7fffffffc720: 0x5564b288      0x00005555      0xf7d4e17a      0x00007fff
0x7fffffffc730: 0x3f800000      0x00000000      0x00000000      0x00010164
0x7fffffffc740: 0x3fc00000      0x00000000      0x00000000      0x00010164
0x7fffffffc750: 0x40000000      0x00000000      0x00000000      0x00010164
0x7fffffffc760: 0x40200000      0x00000000      0x00000000      0x00010164
0x7fffffffc770: 0x40400000      0x00000000      0x00000000      0x00010164
0x7fffffffc780: 0x00000000      0x40000000      0x00000000      0x00020296
0x7fffffffc790: 0x00000000      0x40200000      0x00000000      0x00020296
0x7fffffffc7a0: 0x00000000      0x40400000      0x00000000      0x00020296
0x7fffffffc7b0: 0x00000000      0x40600000      0x00000000      0x00020296
0x7fffffffc7c0: 0x00000000      0x40800000      0x00000000      0x00020296
0x7fffffffc7d0: 0x00000000      0x00000000      0x40400000      0x000303c8
0x7fffffffc7e0: 0x00000000      0x00000000      0x40600000      0x000303c8
0x7fffffffc7f0: 0x00000000      0x00000000      0x40800000      0x000303c8
0x7fffffffc800: 0x00000000      0x00000000      0x40900000      0x000303c8
0x7fffffffc810: 0x00000000      0x00000000      0x40a00000      0x000303c8
0x7fffffffc820: 0xffffc890      0x00007fff      0x55666660      0x00005555
0x7fffffffc830: 0xffffc850      0x00007fff      0xfae82b00      0xf4f52495
0x7fffffffc840: 0xffffc890      0x00007fff      0x556d76a8      0x00005555
0x7fffffffc850: 0xffffc870      0x00007fff      0x5556f17b      0x00005555
```

Figure 11: Stack state at Breakpoint 2 after memcpy() execution in try_combine() during normal operation.

This analysis established that the system's stack and register states were initially consistent with nominal execution showing no evidence of memory corruption.

4.1.2 Exploitation state

During the execution of the try_combine() function before the attack is launched, the stack frame still conforms to the standard x86_64 calling conventions. All the parameters conform to the normal operation state with the saved rip = 0x55555556f17b located at address 0x7fffffffc858.

However, when memcpy() is invoked with a buffer_offset exceeding 256 bytes, the operation writes beyond the bounds of unsafe_buffer, thereby corrupting stack memory above it. This overwrite extends into the control data region and modifies the contents at 0x7fffffffc858, resulting in the saved rip being overwritten with 0x2029600000000 a new address which does not correspond to the intended return location.

```
Before memcpy
After memcpy

Thread 1 "concatenator_no" hit Breakpoint 2, PointCloudConcatenator::try_combine (
    ↪ this=0x5555556662c0)
    at /home/user/ros2_ws/src/my_pkg/src/pointcloud_concatenator.cpp:70
70          combined_cloud->data.assign(unsafe_buffer, unsafe_buffer +
    ↪ buffer_offset);

(gdb) info frame
Stack level 0, frame at 0x7fffffffc860:
 rip = 0x55555556fec4 in PointCloudConcatenator::try_combine (/home/user/ros2_ws/
    ↪ src/my_pkg/src/pointcloud_concatenator.cpp:70);
    saved rip = 0x2029600000000
 called by frame at 0x7fffffffc868
 source language c++.
 Arglist at 0x7fffffffc850, args: this=0x5555556662c0
 Locals at 0x7fffffffc850, Previous frame's sp is 0x7fffffffc860
 Saved registers:
  rbx at 0x7fffffffc848, rbp at 0x7fffffffc850, rip at 0x7fffffffc858
```

Figure 12: Stack corruption after memcpy() execution in try_combine() showing saved return address overwrite.

This corrupted return address ultimately causes control to jump to an unintended location upon function return which violates the program execution flow leading to stack corruption ("stack smashing detected") and terminates with SIGABRT, ultimately affecting the lidar data fusion.

```
(gdb) bt
#0  PointCloudConcatenator::try_combine (this=0x5555556662c0) at /home/user/
    ↪ ros2_ws/src/my_pkg/src/pointcloud_concatenator.cpp:70
#1  0x0002029600000000 in ?? ()
#2  0x40d0000000000000 in ?? ()
#3  0x0002029600000000 in ?? ()
#4  0x0000000000000000 in ?? ()
(gdb) continue
Continuing.
*** stack smashing detected ***: terminated

Thread 1 "concatenator_no" received signal SIGABRT, Aborted.
__pthread_kill_implementation (no_tid=0, signo=6, threadid=140737350188864) at ./
    ↪ nptl/pthread_kill.c:44
44      ./nptl/pthread_kill.c: No such file or directory.
```

Figure 13: Control flow disruption after memcpy() in try_combine() due to return address corruption.

This exploitation shows that the rip can be manipulated and set to an invalid address. It demonstrates the potential for arbitrary code execution in CAV systems. These results validate the risks caused by buffer overflows in real-time lidar processing, where unchecked data inputs lead to critical failures in the perception module.

4.2 Performance Evaluation of the Virtualized Security Framework

This section outlines the results of evaluating the Virtualized Security framework. We used a python controller to switch among different Virtual Network Functions (VNFS) that implement a point cloud concatenation application. The controller uses a continuous switching mechanism to orchestrate the three VNFs from C++ VNF(also stated as cpp-VNF), Python VNF, and Java VNF, where each VNF is allotted a fixed 5-second cycle. During its turn, the controller first ensures the VNF's port is free by terminating any stale instances. It then launches a new instance in a non-blocking manner using SSH and background execution, without waiting for the VNF to fully initialize.

After a 2-second delay, it performs a quick liveness check by querying the process ID and inspecting log files. The controller immediately proceeds to the next VNF when the 5-second interval expires. This non-blocking design, coupled with the fixed switching cycle ensures that system execution continues uninterrupted, as any attack targeting a specific programming language VNF is isolated to 5-second slot, allowing the controller to swiftly transition to a different VNF and maintain overall service continuity.

4.2.1 Code Diversification Results

During the normal runtime, the C++ concatenator in the cpp-vnf receives three 48-byte point-cloud fragments (3 points each), combines them, and publishes 9 points (144 bytes) well below its 256-byte stack buffer and no overflow occurs. The controller then switches to the python-vnf where the python concatenator receives the same three fragments, appends the 144-byte cloud data, and publishes it. Five seconds later java-vnf also contains the java concatenator that performs the same 144-byte aggregation and relays it unchanged. Therefore, the logs show smooth operation during normal behavior.

When the point-cloud publishers send large payloads (e.g.1600 bytes), the C++ concatenator receives excess bytes beyond its fixed buffer size of only 256 bytes. This leads to a stack buffer overflow, causing the process to crash ("Segmentation fault (core dumped)"). As the controller switches to Python VNF, the point cloud concatenation is successfully conducted without any interruption even in the presence of excess point cloud data. Even when the controller switches to Java VNF, it remains unaffected by the large payloads, confirming the effectiveness of the proposed code diversification approach.

```
[2025-06-16 00:19:01] cpp-vnf     10.20.20.14
[2025-06-16 00:19:01] started.
5562/tcp:   154766
[2025-06-16 00:19:14] pid: 155137
[2025-06-16 00:19:14] last log lines:
Waiting for message on SUB[1]...
 Received 32 bytes on SUB[1]  (~2 points)
 Waiting for message on SUB[2]...
 Received 32 bytes on SUB[2]  (~2 points)
Safe: combined size 96 fits <256 bytes
[2025-06-16 00:19:14] sleep 5s

[2025-06-16 00:19:19] python-vnf     10.20.20.26
[2025-06-16 00:19:19] started.
[2025-06-16 00:19:33] pid: 423150
[2025-06-16 00:19:33] last log lines:
Published combined point cloud of size 96
Published combined point cloud of size 96
Published combined point cloud of size 96
Published combined point cloud of size 96
Published combined point cloud of size 96
[2025-06-16 00:19:33] sleep 5s

[2025-06-16 00:19:38] java-vnf     10.20.20.113
[2025-06-16 00:19:38] started.
[2025-06-16 00:19:51] pid: 146478
[2025-06-16 00:19:51] last log lines:
Published combined point cloud of size 96
Published combined point cloud of size 96
Published combined point cloud of size 96
Published combined point cloud of size 96
Published combined point cloud of size 96
[2025-06-16 00:19:51] sleep 5s
```

Figure 14: Log trace of code diversification (normal point cloud data).

```
[2025-06-16 12:47:32] cpp-vnf     10.20.20.14
[2025-06-16 12:47:32] started.
bash: line 1: 181503 Segmentation fault      (core dumped)
[2025-06-16 12:47:44] pid: 181555
[2025-06-16 12:47:44] last log lines:
(empty)
[2025-06-16 12:47:44] sleep 5s

[2025-06-16 12:47:50] python-vnf     10.20.20.26
[2025-06-16 12:47:50] started.
 5412445563/tcp:
[2025-06-16 12:48:02] pid: 541685
[2025-06-16 12:48:02] last log lines:
Published combined point cloud of size 48000
Published combined point cloud of size 48000
Published combined point cloud of size 48000
Published combined point cloud of size 48000
Published combined point cloud of size 48000
[2025-06-16 12:48:03] sleep 5s

[2025-06-16 12:48:08] java-vnf     10.20.20.113
[2025-06-16 12:48:08] started.
[2025-06-16 12:48:21] pid: 186186
[2025-06-16 12:48:21] last log lines:
Published combined point cloud of size 48000
Published combined point cloud of size 48000
Published combined point cloud of size 48000
Published combined point cloud of size 48000
Published combined point cloud of size 48000
[2025-06-16 12:48:21] sleep 5s
```

Figure 15: Log trace of code diversification (excess point cloud data)

We also measured attack success rate by varying the switching interval from 5 to 30 seconds. To evaluate this metric, we implemented a hybrid version of our code diversification approach. The hybrid version includes both proactive switching with a fixed interval and reactive switching that triggers the switch when an attack is detected. Our evaluation scenario involves a legitimate point cloud publisher and a malicious publisher. We considered the same point cloud concatenation application as described above. The malicious publisher was configured to send large payloads at 10 random time instants during a 2-minute period. Figure 16 shows the attack success rate for different switching intervals. We observe that as the switching interval increases from 5 to 15 seconds, the attack success rate decreases significantly from 32% to 18%. The smallest switching interval (5sec) results in too frequent switching from one VNF to another, thereby enabling the hacker to repeatedly access the same vulnerable C++ VNF instance, increasing the probability of a successful buffer overflow exploitation. However, the attack success rate remains constant at intervals of 15, 20, and 25 seconds, with only a marginal decrease observed for a 30-second interval. Our results show the effectiveness of the code diversification approach in achieving up to 84% resiliency through rotating the VNFs.

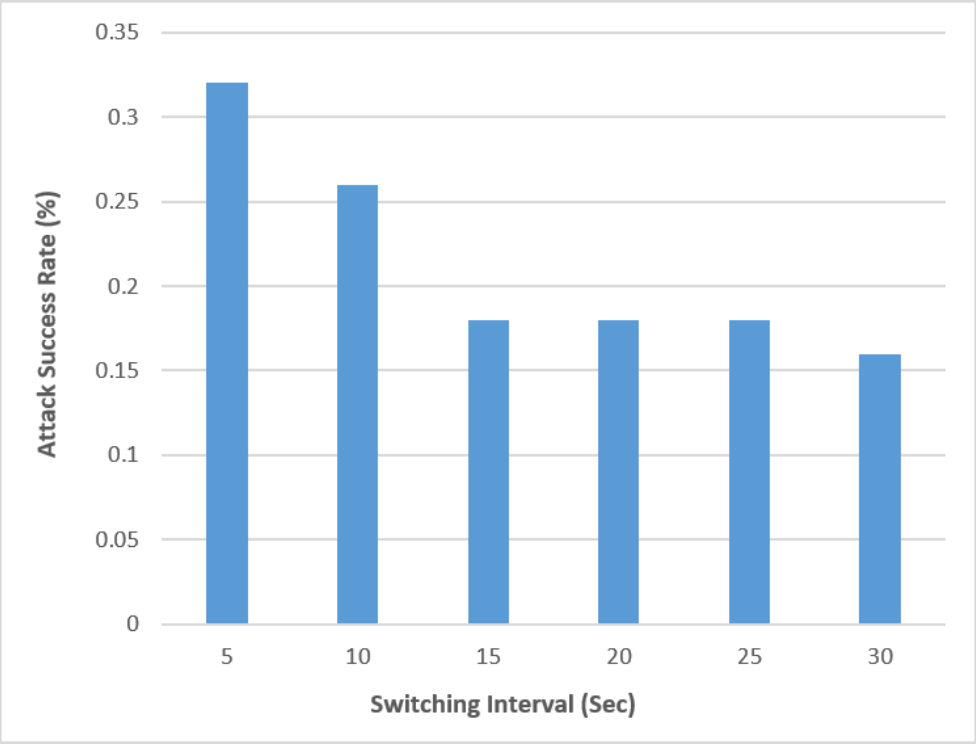

Figure 16: Attack Success Rate.

4.2.2 Resource Usage

Figure 17 shows the CPU usage of the code diversification controller. We observe that the CPU usage (i.e., number of CPU cores) varies over time. During our observation period (300sec), the usage reached around 28 cores. The number of CPU cores stays below 25 for the most part of the observation period. Figure 18 shows memory usage of the controller. We observe low memory usage (10%) during the entire period. It shows the controller has a light workload and has a lower memory requirement.

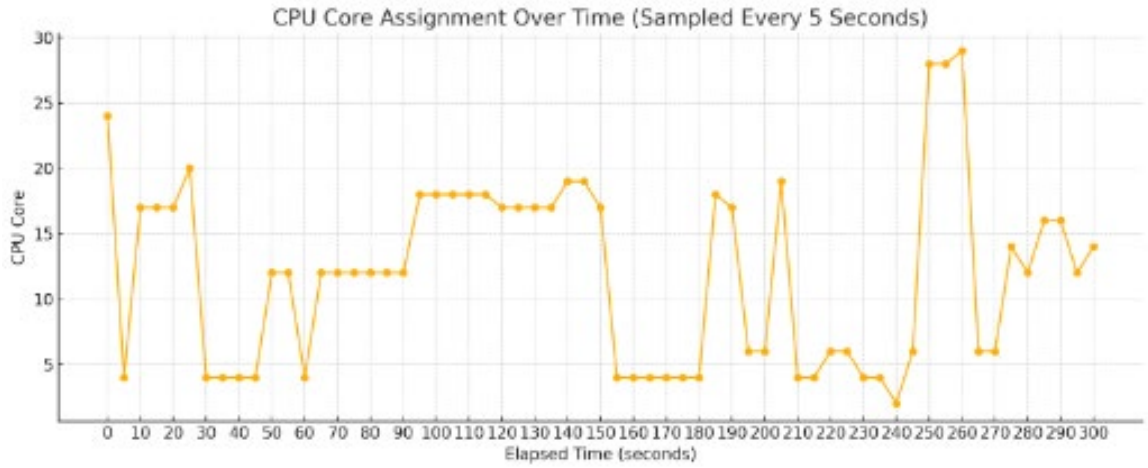

Figure 17: CPU usage for the controller.

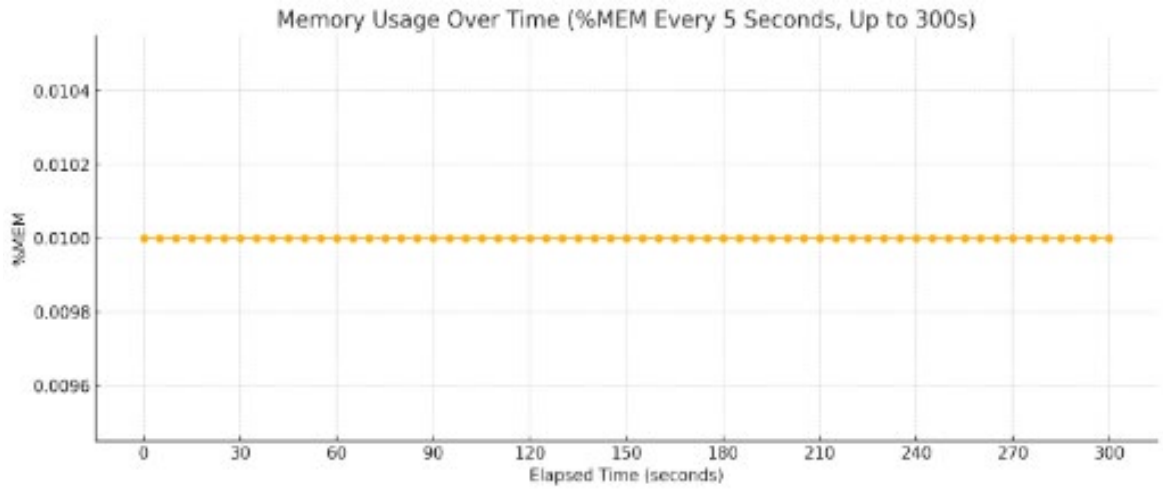

Figure 18: Memory Usage for the Controller.

Figure 19 and Figure 20 show user-level (%usr) and system-level (%sys) CPU utilization for the C++ VNF. We observe that the CPU overhead at the user-level dominates system-level overhead. This is because user-level includes processing the application logic for the point cloud concatenation program that involves memcpy, an expensive operation that copies the received point cloud into memory. %sys has frequent small spikes throughout the entire observation period because the kernel is constantly busy in handling MQTT (exchanging network messages with the MQTT broker) operations, SSH calls from the controller, and socket operations. %usr stays mostly flat with only three clear spikes in CPU utilization. It may refer to the execution of the memcpy operation. We observe that both metrics peak simultaneously at t=145s (%usr=4.0%, %sys=1.3%), confirming the processing of point cloud data. As the memcpy function is executed at the application-level, the C++ VNF consumes more CPU at the user-level than the kernel. The periods of zero CPU utilization, both at the user-level and system-level, observed in the C++ VNF correspond to intervals during which the controller has switched to another VNF to run the point cloud concatenation application.

The CPU utilization at the user-level (shown in Figure 21) for Java VNF shows similar trends to the C++ VNF. The Java VNF consumes more CPU at the user-level than at the kernel level. As the controller switches between VNFs to create diversity in the attack surface, the Java VNF remains idle and reports zero CPU utilization during certain intervals of the observation period. Java VNF reports slightly higher average CPU utilization both at the user-level and system level than the C++ VNF. The reason for higher %usr is that the Java applications run on the Java Virtual Machine (JVM), which introduces additional user-space processing. The higher %sys is reported because the JVM performs more system-level operations (e.g., MQTT communications and socket operations) compared to a C++ application. Figure 23 and Figure 24 show the memory usage reported by C++ and Java VNF, respectively. The memory usage of C++ VNF stays within the range ~ (202 MiB -217MiB) while Java VNF uses memory in the range ~(192 MiB – 242 MiB). On average, the memory usage of the Java VNF and the C++ VNF remains nearly identical during the observation period, with average values of 210 MB and 211 MB, respectively.

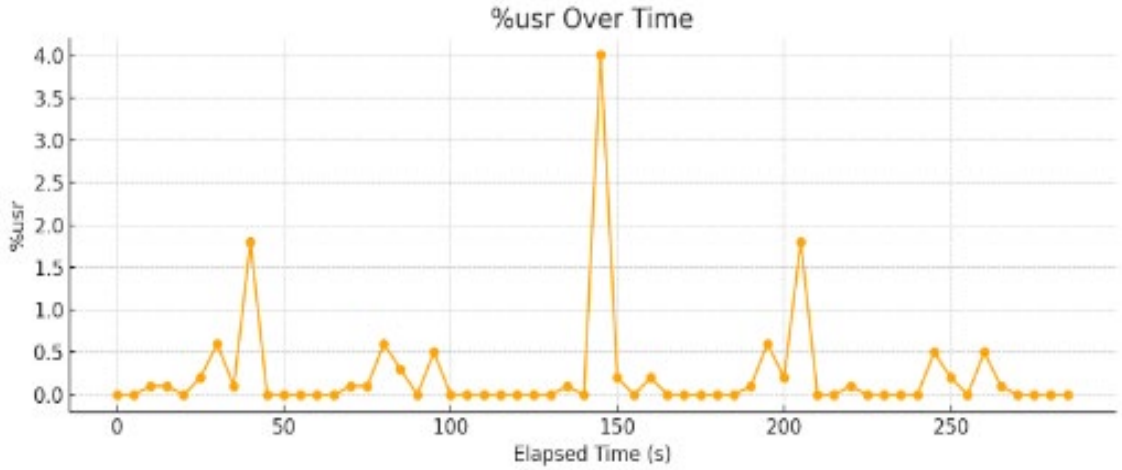

Figure 19: CPU usage % usr for the C++ VNF.

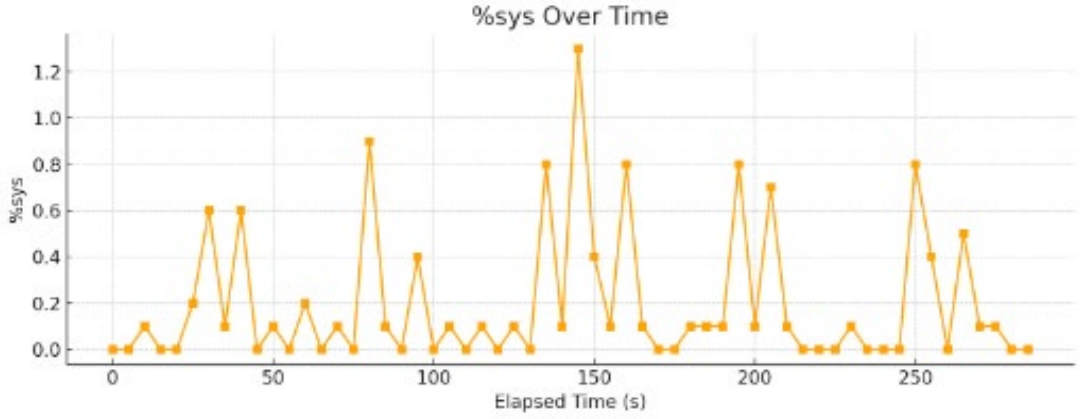

Figure 20: CPU usage % sys for the C++ VNF.

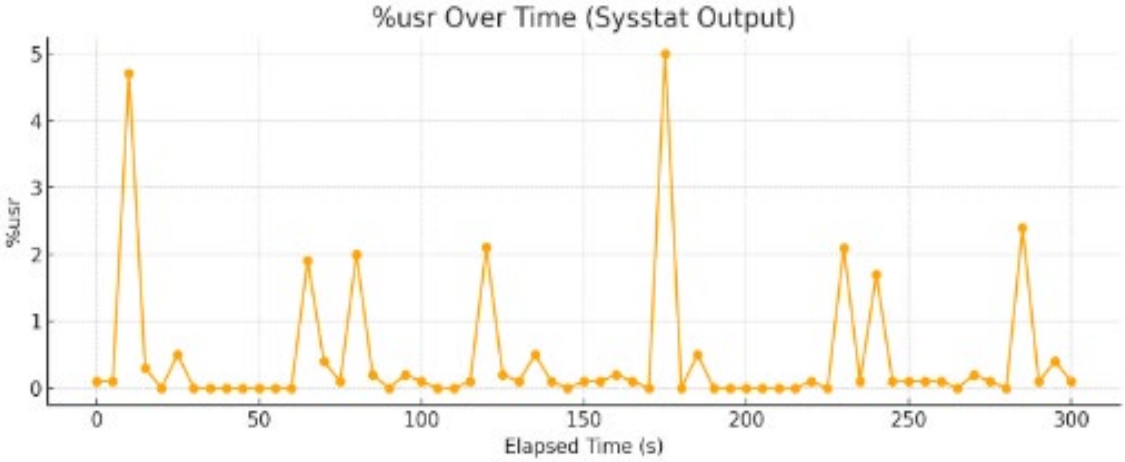

Figure 21: CPU usage % usr for the Java VNF.

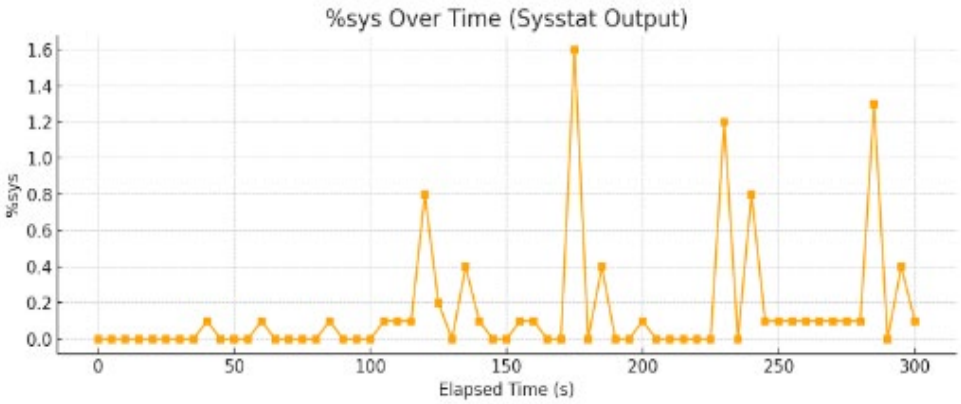

Figure 22: CPU usage % sys for the Java VNF.

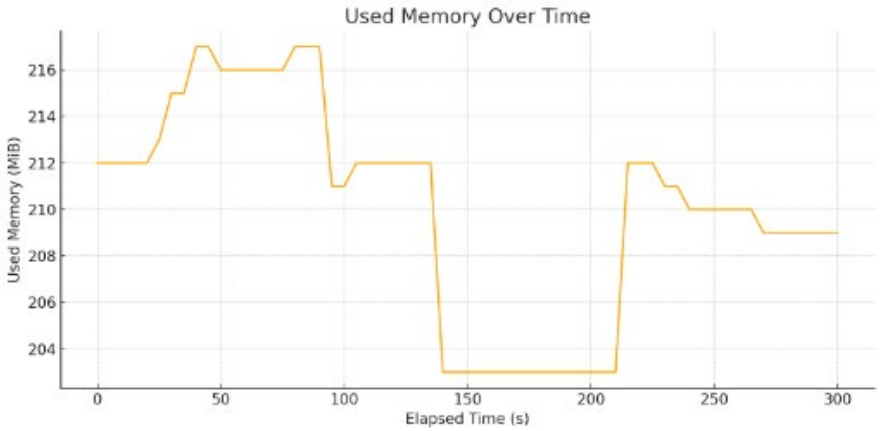

Figure 23: Memory Usage for the C++ VNF.

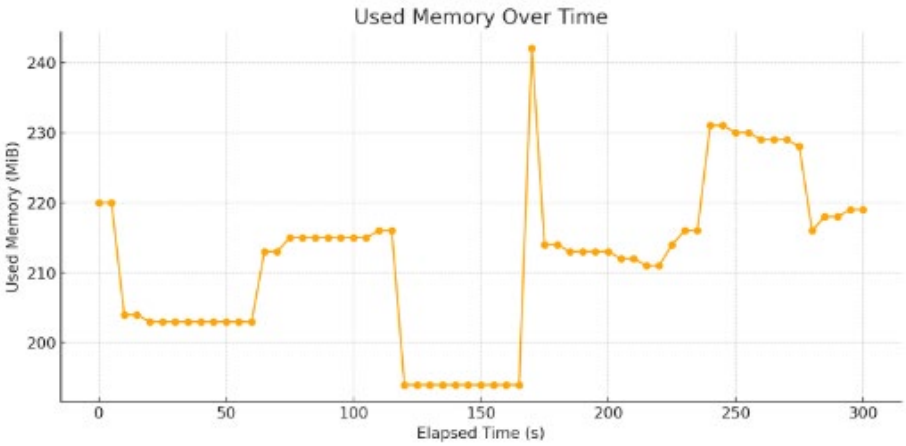

Figure 24: Memory Usage for the Java VNF.

4.3 Performance Evaluation of the DDQN Agent

This section presents the results of evaluating the proposed adaptive defense mechanism driven by the DDQN agent. The evaluation was conducted within a threat simulation environment intended to mirror real-world software vulnerability risks and adversarial behaviors. To assess the performance of the DDQN agent, a traditional Q-Learning baseline was included for comparative analysis, with both agents exposed to identical environment.

Table 1 summarizes two metrics: QoS (Uptime) and MTBI achieved by DDQN and QL. We observe that DDQN outperforms the QL in terms of both metrics. DDQN achieves a QoS of 98.66%, whereas QL shows 97.62% QoS. It shows the ability of DDQN in ensuring continuity of the CAV services. We also observe that DDQN achieves higher MTBI (11 steps) compared to QL (9.4 steps), proving the enhanced system resilience offered by DDQN.

Table 5. Performance summary (DDQN vs Q-Learning).

Metric	DDQN Agent	Q-Learning Agent
QoS (Uptime) (%)	98.66	97.62
MTBI (steps)	11.0	9.4

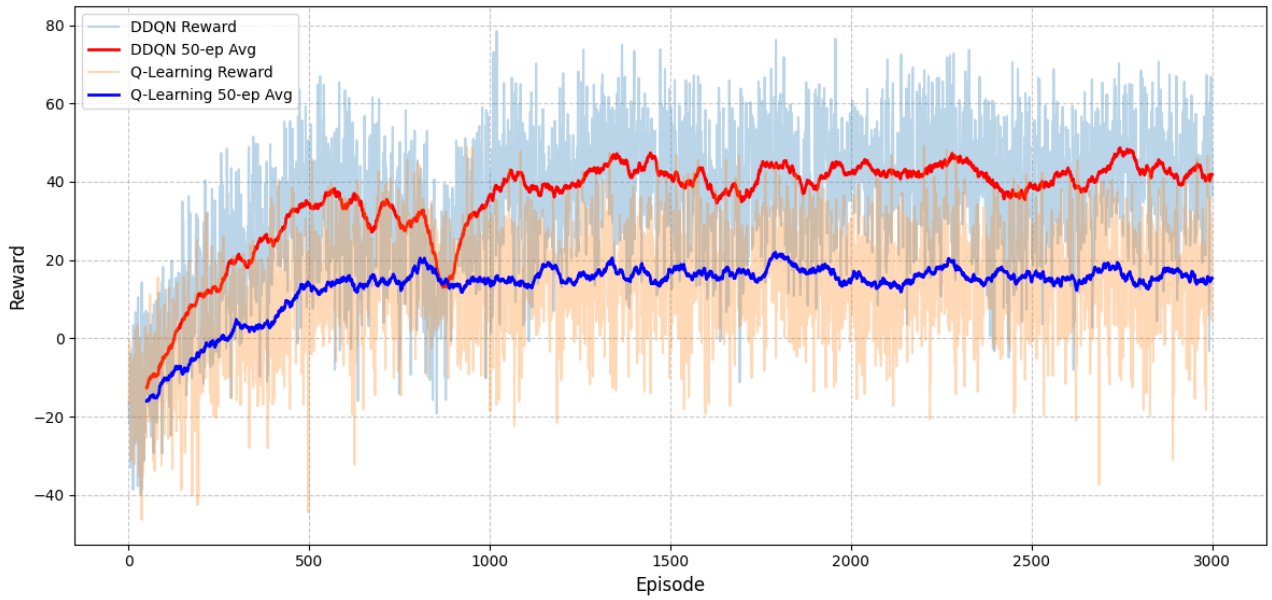

Figure 25: Reward Progression.

Figure 25 shows the raw and smoothed (50-episode moving average) reward obtained by the DDQN and Q-learning agent. We observe that the DDQN agent exhibits more stable and higher rewards over time compared to Q-Learning. Both DDQN and the Q-learning agent started with an identical reward. However, the reward of the DDQN agent grows faster compared to the Q-learning agent, thereby confirming the ability of the DDQN agent in learning the optimal policy in an efficient manner.

Figure 26 shows the cumulative average reward of both agents. We observe that DDQN consistently outperforms Q-Learning over time, indicating better policy convergence and long-term learning behavior. The reward curves clearly demonstrate a stable convergence for the DDQN agent. The reward gap between DDQN and Q-learning gradually increases over time, reflecting Q-learning's limited scalability in high-dimensional state spaces and vulnerability to the curse of dimensionality. This disparity reinforces the role of deep function approximators like DDQN in learning adaptive and robust security policies. Figure 27 shows the security success rate achieved by the DDQN agent and the Q-learning agent. The DDQN agent maintains a higher SSR (close to 92%) than Q-Learning (around 89.5%) during all episodes, demonstrating improved defense response by DDQN.

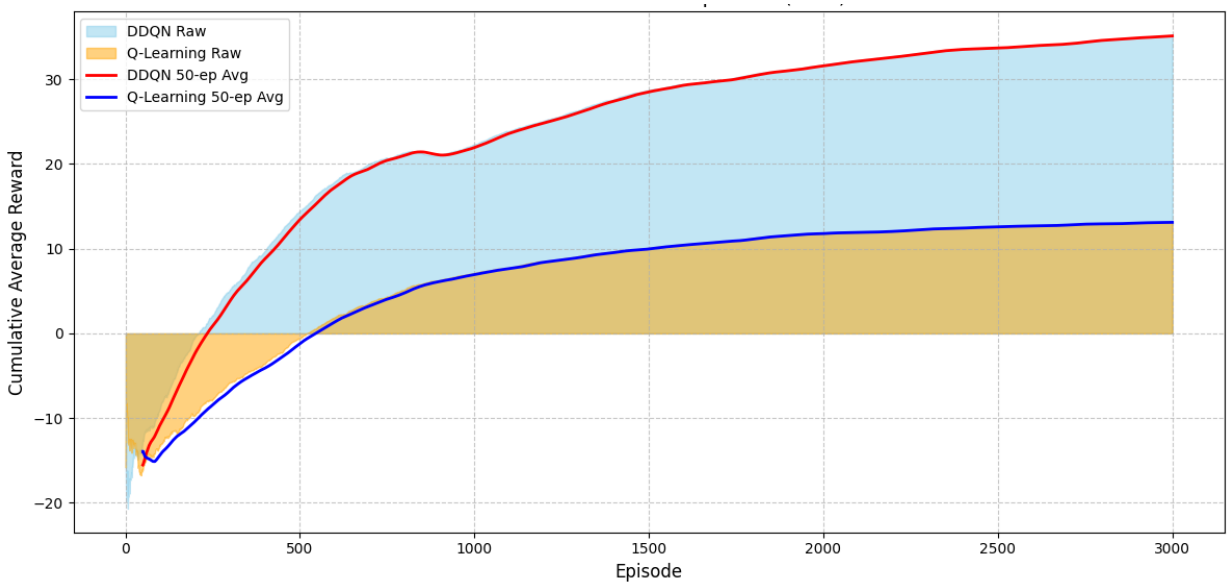

Figure 26: Cumulative average reward progression.

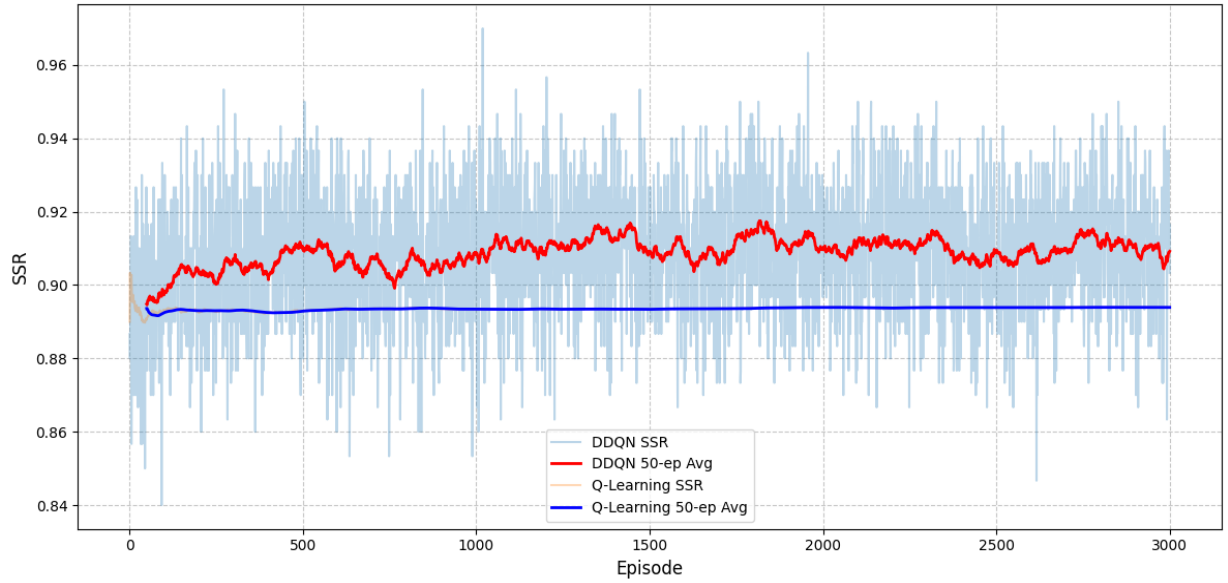

Figure 27: Security Success Rate (SSR).

CHAPTER 5

Conclusions & Future Works

5.1 Summary

This study demonstrates a lidar buffer overflow exploitation in CAV software by using a vulnerable point cloud concatenation node that causes memory corruption. The results show the risks posed by inadequate memory management in a real-time sensor fusion process where manipulated lidar data can destabilize the perception module. This study developed a robust and adaptive code diversification-based defense mechanism for CAVs. The defense mechanism leverages the principles of MTD and applies a DDQN agent to learn the optimal policy. By intelligently and actively switching to lower-risk code variants based on the dynamic state assessment, the mechanism shows a Security Success Rate (SSR) of over 98% and maintains a perfect QoS of 98.66%. This study also proposed an NFV-based virtualized security framework to facilitate the dynamic execution of the code variants such as VNFs. The framework allows scalability, elasticity, and reduced management effort and operational cost due to the NFV paradigm. Our prototype-based evaluation shows the effectiveness of code diversification in preventing cyber-attacks by switching among the VNFs.

5.2 Future Research Directions

While this study provides insightful insights, some limitations must be acknowledged: The attack exploitation focused only on lidar buffer overflows, leaving other sensor modalities unexplored. The AWSIM-Autoware environment cannot fully replicate the complexities of real-world driving environments and hardware vulnerabilities. To address these limitations and extend the impact of this research, future work should focus on evaluating other memory attack vectors, including radar, camera-based and V2X protocol exploits, and integrating hardware-in-the-loop (HIL) testing to validate findings under more diverse and realistic conditions. Future works will also include investigating game-theoretic models that involve formulating the code diversification scenario as Stackelberg games to simulate dynamic attacker-defender interactions in a more practical manner. We will also investigate Multi-Agent Reinforcement Learning (MARL) techniques to deploy cooperative and competitive agents to handle multi-point cyberattacks across different CAV subsystems (e.g., Perception, Navigation, Diagnostics) simultaneously.

REFERENCES

Andreou, A., Constandinos, ·, Mavromoustakis, X., Markakis, E., Athina Bourdena, ·, & Mastorakis, G. (2025). Enhancing network slice security with Deep Reinforcement Learning and Moving Target Defense strategies. *Discover Internet of Things 2025 5:1*, *5*(1), 1–17. https://doi.org/10.1007/S43926-025-00161-1

AWSIM document. (n.d.). Retrieved June 15, 2025, from https://tier4.github.io/AWSIM/

Benibo, I., Sahoo, J., Mwakalonge, J., Gyimah, N. K., Comert, G., Biswal, B., & Swain, N. (2025). Q-Learning-Based Adaptive Defense Mechanism for Connected Autonomous Vehicles. *Conference Proceedings - IEEE SOUTHEASTCON*, 1063–1064. https://doi.org/10.1109/SOUTHEASTCON56624.2025.10971541

Cengiz, E., & Gök, M. (2023). Reinforcement Learning Applications in Cyber Security: A Review. *Sakarya University Journal of Science*, *27*(2), 481–503. https://doi.org/10.16984/SAUFENBILDER.1237742

Cho, J.-H., Sharma, D. P., Alavizadeh, H., Yoon, S., Ben-Asher, N., Moore, T. J., Kim, D. S., Lim, H., & Nelson, F. F. (2020). Toward Proactive, Adaptive Defense: A Survey on Moving Target Defense. *IEEE Communications Surveys & Tutorials*, *22*(1), 709–745. https://doi.org/10.1109/COMST.2019.2963791

Coronado, E., Cebrián-Márquez, G., & Riggio, R. (2020). Enabling Autonomous and Connected Vehicles at the 5G Network Edge. *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, 350–352. https://doi.org/10.1109/NetSoft48620.2020.9165444

Eiza, M. H., & Ni, Q. (2017). Driving with Sharks: Rethinking Connected Vehicles with Vehicle Cybersecurity. *IEEE Vehicular Technology Magazine*, *12*(2), 45–51. https://doi.org/10.1109/MVT.2017.2669348

Elkhail, A. A., Refat, R. U. D., Habre, R., Hafeez, A., Bacha, A., & Malik, H. (2021). Vehicle Security: A Survey of Security Issues and Vulnerabilities, Malware Attacks and Defenses. *IEEE Access*, *9*, 162401–162437. https://doi.org/10.1109/ACCESS.2021.3130495

ETSI. (2021). *NFV Release 2 Description*.

Farris, I., Taleb, T., Khettab, Y., & Song, J. (2019). A Survey on Emerging SDN and NFV Security Mechanisms for IoT Systems. *IEEE Communications Surveys & Tutorials*, *21*(1), 812–837. https://doi.org/10.1109/COMST.2018.2862350

Flawfinder Home Page. (n.d.). Retrieved June 15, 2025, from https://dwheeler.com/flawfinder/

Goldman Sachs. (2022, November). *Software Is Taking Over the Auto Industry*.

Halat, S., Ebadzadeh, M. M., & Amani, K. (2024). Modified Double-DQN: addressing stability. *11th International Symposium on Telecommunication: Communication in the Age of Artificial Intelligence, IST 2024*, 697–702. https://doi.org/10.1109/IST64061.2024.10843545

Hataba, M., Sherif, A., & Elkhouly, R. (2021a). A Proposed Software Protection Mechanism for Autonomous Vehicular Cloud Computing. *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 878–881. https://doi.org/10.1109/MWSCAS47672.2021.9531682

Hataba, M., Sherif, A., & Elkhouly, R. (2021b). A Proposed Software Protection Mechanism for Autonomous Vehicular Cloud Computing. *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 878–881. https://doi.org/10.1109/MWSCAS47672.2021.9531682

Hataba, M., Sherif, A., & Elkhouly, R. (2022a). Enhanced Obfuscation for Software Protection in Autonomous Vehicular Cloud Computing Platforms. *IEEE Access*, *10*, 33943–33953. https://doi.org/10.1109/ACCESS.2022.3159249

Hataba, M., Sherif, A., & Elkhouly, R. (2022b). Enhanced Obfuscation for Software Protection in Autonomous Vehicular Cloud Computing Platforms. *IEEE Access*, *10*, 33943–33953. https://doi.org/10.1109/ACCESS.2022.3159249

Hosseinzadeh, S., Rauti, S., Laurén, S., Mäkelä, J. M., Holvitie, J., Hyrynsalmi, S., & Leppänen, V. (2018). Diversification and obfuscation techniques for software security: A systematic literature review. *Information and Software Technology*, *104*, 72–93. https://doi.org/10.1016/J.INFSOF.2018.07.007

Jadhav, S., Sonwalkar, V., Shewale, S., Shitole, P., Bhujadi, S., & Pate, G. K. (n.d.). *Deep Reinforcement Learning for Autonomous Driving Systems*. www.ijfmr.com

Jangda, A., Mishra, M., & De Sutter, B. (2015). Adaptive just-in-time code diversification. *MTD 2015 - Proceedings of the 2nd ACM Workshop on Moving Target Defense, Co-Located with: CCS 2015*, 49–53. https://doi.org/10.1145/2808475.2808487

Khlifi, A., Othmani, M., & Kherallah, M. (2025). A Novel Approach to Autonomous Driving Using Double Deep Q-Network-Bsed Deep Reinforcement Learning. *World Electric Vehicle Journal 2025, Vol. 16, Page 138*, *16*(3), 138. https://doi.org/10.3390/WEVJ16030138

Liu, Y., Yang, T., Tian, L., & Pei, J. (2025). SGD-TripleQNet: An Integrated Deep Reinforcement Learning Model for Vehicle Lane-Change Decision. *Mathematics 2025, Vol. 13, Page 235*, *13*(2), 235. https://doi.org/10.3390/MATH13020235

Mahmood, K., & Shila, D. M. (2016). Moving target defense for Internet of Things using context aware code partitioning and code diversification. *2016 IEEE 3rd World Forum on Internet of Things, WF-IoT 2016*, 329–330. https://doi.org/10.1109/WF-IOT.2016.7845457

MartinBob. (2019). Common Vulnerabilities Enumeration (CVE), Common Weakness Enumeration (CWE), and Common Quality Enumeration (CQE). *ACM SIGAda Ada Letters*, *38*(2), 9–42. https://doi.org/10.1145/3375408.3375410

Muncaster, P. (2025, January). *Subaru Bug Enabled Remote Vehicle Tracking and Hijacking*.

Nguyen, T. T., & Reddi, V. J. (2023). Deep Reinforcement Learning for Cyber Security. *IEEE Transactions on Neural Networks and Learning Systems*, *34*(8), 3779–3795. https://doi.org/10.1109/TNNLS.2021.3121870

Openstack. (n.d.). *Openstack*. Retrieved June 18, 2025, from https://www.openstack.org/

Parekh, D., Poddar, N., Rajpurkar, A., Chahal, M., Kumar, N., Joshi, G. P., & Cho, W. (2022). A Review on Autonomous Vehicles: Progress, Methods and Challenges. *Electronics (Switzerland)*, *11*(14). https://doi.org/10.3390/electronics11142162

Potteiger, B., Zhang, Z., Cheng, L., & Koutsoukos, X. (2022). A Tutorial on Moving Target Defense Approaches Within Automotive Cyber-Physical Systems. *Frontiers in Future Transportation*, *Volume 2-2021*. https://doi.org/10.3389/ffutr.2021.792573

Rana, M. M., & Hossain, K. (2023). Connected and Autonomous Vehicles and Infrastructures: A Literature Review. *International Journal of Pavement Research and Technology*, *16*(2), 264–284. https://doi.org/10.1007/S42947-021-00130-1/METRICS

Styugin, M., Zolotarev, V., Prokhorov, A., & Gorbil, R. (n.d.). *New Approach To Software Code Diversification In Interpreted Languages Based On The Moving Target Technology*.

Styugin, M., Zolotarev, V., Prokhorov, A., & Gorbil, R. (2016). New approach to software code diversification in interpreted languages based on the moving target technology. *2016 IEEE 10th International Conference on Application of Information and Communication Technologies (AICT)*, 1–5. https://doi.org/10.1109/ICAICT.2016.7991694

Taguinod, M., Doupé, A., Zhao, Z., & Ahn, G.-J. (2015). Toward a Moving Target Defense for Web Applications. *2015 IEEE International Conference on Information Reuse and Integration*, 510–517. https://doi.org/10.1109/IRI.2015.84

The Autoware Foundation · GitHub. (n.d.). Retrieved June 15, 2025, from https://github.com/autowarefoundation

Tsoupidi, R. M., Troubitsyna, E., & Papadimitratos, P. (2023). Thwarting code-reuse and side-channel attacks in embedded systems. *Computers and Security*, *133*. https://doi.org/10.1016/j.cose.2023.103405

van Hasselt, H., Guez, A., & Silver, D. (2015). *Deep Reinforcement Learning with Double Q-learning*. http://arxiv.org/abs/1509.06461

Yi, J., Chen, L., Zhang, H., Li, Y., & Zhao, H. (2020). A Security Model and Implementation of Embedded Software Based on Code Obfuscation. *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 1606–1613. https://doi.org/10.1109/TrustCom50675.2020.00222

Zhu, M., Cao, J., Cai, Z., He, Z., & Xu, M. (2016). Providing flexible services for heterogeneous vehicles: an NFV-based approach. *IEEE Network*, *30*(3), 64–71. https://doi.org/10.1109/MNET.2016.7474346

Appendix A: Controller Script A

```
#!/usr/bin/env python3
import time
import subprocess
import os
from datetime import datetime
#  GLOBAL CONFIG
SSH_KEY = os.path.expanduser("~/.ssh/vnfkey")
SSH_OPTS = [
    "-o", "BatchMode=yes",
    "-o", "StrictHostKeyChecking=accept-new",
    "-o", "ConnectTimeout=5",
    "-n",
    "-f",]
INTERVAL = 5
LOG_DIR = "~/scripts"
PUBLISHER = {
    "name": "publisher",
    "ip":   "10.20.20.26",  # python-vnf
    "cmd":  "python3 -u ~/publisher.py",
    "port": None,}
AGGREGATORS = [{
        "name": "cpp-vnf",
        "ip":   "10.20.20.14",
        "cmd":  "~/pointcloud_concatenator",
        "port": 5562,},{
        "name": "python-vnf",
        "ip":   "10.20.20.26",
        "cmd":  "python3 -u ~/diversification.py",
        "port": 5563,},{
        "name": "java-vnf",
        "ip":   "10.20.20.113",
        "cmd":  "java -cp ~/jeromq-0.5.2.jar:. PointCloudConcatenator",
        "port": 5560,}]
def ssh_run(ip: str, remote_cmd: str, *, capture=False):
    cmd = ["ssh", *SSH_OPTS, "-i", SSH_KEY, f"ubuntu@{ip}", remote_cmd]
    return subprocess.run(cmd, capture_output=capture, text=True)
def free_port(ip: str, port: int | None):
    if port:
        ssh_run(ip, f"fuser -k {port}/tcp || true && sleep 1")
def start_vnf(vnf: dict, kill_old: bool) -> bool:
    ip, cmd = vnf["ip"], vnf["cmd"]
    logfile = f"{LOG_DIR}/{vnf['name']}.log"
    if kill_old:
        ssh_run(ip, f"pkill -f '{cmd}' || true")
        free_port(ip, vnf["port"]
    remote = f"mkdir -p {LOG_DIR} && nohup {cmd} > {logfile} 2>&1 < /dev/null &"
    print(f"\n[{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}] {vnf['name']}     {ip}")
    ssh_run(ip, remote)
    print(f"[{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}] started.")
    time.sleep(2)
    pid = (
        ssh_run(ip, f"pgrep -f '{cmd}' | head -n1", capture=True)
        .stdout.strip() or "n/a")
    time.sleep(10)  # Give process time to generate output
    tail = (
        ssh_run(ip, f"tail -n 5 {logfile} || true", capture=True)
        .stdout.strip() or "(empty)")
    print(f"[{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}] pid: {pid}")
    print(f"[{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}] last log lines:\n{tail}")
    alive = False
    if pid != "n/a":
        alive = ssh_run(ip, f"ps -p {pid}", capture=True).returncode == 0
    crashed = (not alive) or ("Segmentation fault" in tail)
    if crashed:
        print(f"[{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}]  {vnf['name']} may have crashed.")
    return crashed
print(f"[{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}] Controller running     switch every {INTERVAL}s")
start_vnf(PUBLISHER, kill_old=False)
while True:
    for v in AGGREGATORS:
        start_vnf(v, kill_old=True)
        print(f"[{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}] sleep {INTERVAL}s\n")
        time.sleep(INTERVAL)
```

Appendix B: Publisher Script

```
#!/usr/bin/env python3
# combined_pc_publisher.py     runs horizontal (X), vertical (Y), depth (Z) publishers, plus
    filename publisher
import time
import struct
import zmq
import threading
def publisher_1():  # Horizontal X-variation     tcp://*:5556
    ctx = zmq.Context()
    sock = ctx.socket(zmq.PUB)
    sock.bind("tcp://*:5556")
    time.sleep(1)
    while True:
        buf = bytearray()
        for i in range(2):
            x = 1.0 + i * 0.5
            y = 0.0
            z = 0.0
            buf += struct.pack('<fffBBH', x, y, z, 100, 1, 1)
        sock.send(buf)
        print("Publisher 1     sent horizontal points")
        time.sleep(0.5)
def publisher_2():  # Vertical Y-variation     tcp://*:5557
    ctx = zmq.Context()
    sock = ctx.socket(zmq.PUB)
    sock.bind("tcp://*:5557")
    time.sleep(1)
    while True:
        buf = bytearray()
        for i in range(2):
            x = 0.0
            y = 2.0 + i * 0.5
            z = 0.0
            buf += struct.pack('<fffBBH', x, y, z, 150, 2, 2)
        sock.send(buf)
        print("Publisher 2     sent vertical points")
        time.sleep(0.5)
def publisher_3():  # Depth Z-variation     tcp://*:5558
    ctx = zmq.Context()
    sock = ctx.socket(zmq.PUB)
    sock.bind("tcp://*:5558")
    time.sleep(1)
    while True:
        buf = bytearray()
        for i in range(2):
            x = 0.0
            y = 0.0
            z = 3.0 + i * 0.5
            buf += struct.pack('<fffBBH', x, y, z, 200, 3, 3)
        sock.send(buf)
        print("Publisher 3     sent depth points")
        time.sleep(0.5)
def filename_publisher():  # Publishes filenames     tcp://*:5559
    ctx = zmq.Context()
    sock = ctx.socket(zmq.PUB)
    sock.bind("tcp://*:5559")
    time.sleep(0.2)
    toggle = True  # Switch between normal and malicious names
    while True:
        filename = "out.txt" if toggle else "out.txt; whoami"
        sock.send_string(filename)
        print(f"Published filename: {filename}")
        toggle = not toggle
        time.sleep(0.5)
if __name__ == '__main__':
    threads = [
        threading.Thread(target=publisher_1, name="pub1"),
        threading.Thread(target=publisher_2, name="pub2"),
        threading.Thread(target=publisher_3, name="pub3"),
        threading.Thread(target=filename_publisher, name="fname-pub"),]
    for t in threads:
        t.start()
    for t in threads:
        t.join()
```

Appendix C: Point Cloud Concatenator: C++ Code

```
// pointcloud_concatenator.cpp (vulnerable)
#include <zmq.hpp>
#include <iostream>
#include <thread>
#include <cstring>
int main() {
  zmq::context_t ctx{1};
  std::vector<zmq::socket_t> subs;
  for (int port : {5556, 5557, 5558}) {
    zmq::socket_t s(ctx, zmq::socket_type::sub);
    std::string address = "tcp://192.168.222.168:" + std::to_string(port);
    std::cout << " Connecting SUB socket to " << address << "\n";
    s.connect(address);
    s.set(zmq::sockopt::subscribe, "");
    subs.push_back(std::move(s));
  }
  zmq::socket_t pub(ctx, zmq::socket_type::pub);
  pub.bind("tcp://*:5562");
  std::cout << "C++ concatenator running (vulnerable)   \n";
  while (true) {
    uint8_t buffer[256];
    size_t offset = 0;
    bool overflow_imminent = false;
    for (size_t i = 0; i < subs.size(); ++i) {
      zmq::message_t m;
      std::cout << " Waiting for message on SUB[" << i << "]...\n";
      auto recv_res = subs[i].recv(m, zmq::recv_flags::none);
      std::cout << " Received " << m.size() << " bytes on SUB[" << i
                << "]  (~" << m.size() / 16 << " points)\n";
      (void)recv_res;
      if (offset + m.size() > sizeof(buffer)) {
        overflow_imminent = true;
      }
      std::memcpy(buffer + offset, m.data(), m.size());
      offset += m.size();
    }

    if (!overflow_imminent) {
      std::cout << "Safe: combined size " << offset
                << " fits within buffer limit " << sizeof(buffer) << "\n";
    } else {
      std::cout << " Buffer overflow: attempted to pack "
                << offset / 16 << " points into space for only "
                << sizeof(buffer) / 16 << " points\n";
    }
    pub.send(zmq::buffer(buffer, offset), zmq::send_flags::none);
    std::cout << " Published combined (" << offset
              << " bytes     " << offset / 16
              << " points) on tcp://*:5562\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
  }
  return 0;
}
```

Appendix D: Point Cloud Concatenator: Python Code

```python
#!/usr/bin/env python3
import zmq
import os

def main():
    ctx = zmq.Context()

    # Subscribe to 3 point cloud streams
    data_ports = [5556, 5557, 5558]
    subs = []
    for p in data_ports:
        s = ctx.socket(zmq.SUB)
        s.connect(f"tcp://localhost:{p}")
        s.setsockopt(zmq.SUBSCRIBE, b"")
        subs.append(s)

    # Subscribe to filename publisher
    fname_sub = ctx.socket(zmq.SUB)
    fname_sub.connect("tcp://localhost:5559")
    fname_sub.setsockopt(zmq.SUBSCRIBE, b"")

    # Publisher for the combined cloud
    pub = ctx.socket(zmq.PUB)
    pub.bind("tcp://*:5563")

    # Setup poller
    poller = zmq.Poller()
    for s in subs:
        poller.register(s, zmq.POLLIN)
    poller.register(fname_sub, zmq.POLLIN)

    pc_bufs = [None] * 3
    filename = None

    while True:
        # Wait until all three point clouds and filename arrive
        while True:
            ready = dict(poller.poll())
            for idx, s in enumerate(subs):
                if s in ready and pc_bufs[idx] is None:
                    pc_bufs[idx] = s.recv()
            if fname_sub in ready and filename is None:
                filename = fname_sub.recv_string()
            if all(pc_bufs) and filename:
                break

        combined = b''.join(pc_bufs)
        print(f"Injected shell input received: {filename}")

        try:
            str = f"echo combined > {filename}"
            os.system(str)
        except Exception as e:
            print(f"Error executing: {e}")

        pub.send(combined)
        print(f"Published combined point cloud of size {len(combined)}")

        pc_bufs = [None] * 3
        filename = None

if __name__ == "__main__":
    main()
```

Appendix E: Point Cloud Concatenator: Java Code

```
import org.zeromq.ZMQ;
import java.util.ArrayList;
import java.util.List;

public class PointCloudConcatenator {
    public static void main(String[] args) {
        ZMQ.Context context = ZMQ.context(1);

        // Sockets to receive from 3 sources
        ZMQ.Socket sub1 = context.socket(ZMQ.SUB);
        ZMQ.Socket sub2 = context.socket(ZMQ.SUB);
        ZMQ.Socket sub3 = context.socket(ZMQ.SUB);

        sub1.connect("tcp://192.168.222.168:5556");
        sub2.connect("tcp://192.168.222.168:5557");
        sub3.connect("tcp://192.168.222.168:5558");

        sub1.subscribe(ZMQ.SUBSCRIPTION_ALL);
        sub2.subscribe(ZMQ.SUBSCRIPTION_ALL);
        sub3.subscribe(ZMQ.SUBSCRIPTION_ALL);

        // Socket to publish concatenated data on 5560
        ZMQ.Socket pub = context.socket(ZMQ.PUB);
        pub.bind("tcp://*:5560");

        System.out.println("Java aggregator started     publishing to port
           5560");

        while (true) {
            byte[] msg1 = sub1.recv();
            byte[] msg2 = sub2.recv();
            byte[] msg3 = sub3.recv();

            List<byte[]> all = new ArrayList<>();
            all.add(msg1);
            all.add(msg2);
            all.add(msg3);

            int totalSize = msg1.length + msg2.length + msg3.length;
            byte[] combined = new byte[totalSize];

            int pos = 0;
            for (byte[] part : all) {
                System.arraycopy(part, 0, combined, pos, part.length);
                pos += part.length;
            }

            pub.send(combined);
            System.out.println("Published combined point cloud of size " +
               combined.length);
        }
    }
}
```

Appendix F: Vulnerabilities in C++ modules (Autoware) detected by Flaw Finder Tool

FILE PATH	LEVEL	VULNERABILITY TYPE	VULNERABILITY DETAILS	VULNERABILITY CODE	MITIGATION
/home/arthur/autoware/src/universe/autoware.universe/control/autoware_vehicle_cmd_gate/test/src/test_filter_in_vehicle_cmd_gate_node.cpp:427: 428 :429	4	shell	This causes a new program to execute and is difficult to use safely	CWE-78	try using a library call that implements the same functionality if available.
/home/arthur/autoware/src/universe/autoware.universe/planning/behavior_path_planner/autoware_behavior_path_start_planner_module/src/start_planner_module.cpp:1857:	4	(format) vsnprintf:	If format strings can be influenced by an attacker, they can be exploited, and note that sprintf variations do not always \0-terminate	CWE-134	Use a constant for the format specification.
/home/arthur/autoware/src/universe/autoware.universe/system/bluetooth_monitor/service/main.cpp:49: [3] (buffer) getopt_long:	3	(buffer)	Some older implementations do not protect against internal buffer overflows	(CWE-120, CWE-20)	Check implementation on installation, or limit the size of all string inputs.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_bytetrack/src/bytetrack_node.cpp:105: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120)	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_compare_map_segmentation/src/distance_based_compare_map_filter/node.cpp:153: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_compare_map_segment	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.

ation/src/distance_based_compare_map_filter/node.cpp:154: [2] (buffer) memcpy:					
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_compare_map_segmentation/src/distance_based_compare_map_filter/node.cpp:155: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_compare_map_segmentation/src/distance_based_compare_map_filter/node.cpp:159: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_compare_map_segmentation/src/voxel_based_approximate_compare_map_filter/node.cpp:122: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_compare_map_segmentation/src/voxel_based_approximate_compare_map_filter/node.cpp:123: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_compare_map_segmentation/src/voxel_based_approximate_compare_map_filter/node.cpp:124: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_compare_map_segment	2	(buffer) memcpy:	Does not check for buffer overflows when copying to	(CWE-120).	Make sure destination can always hold the source data.

ation/src/voxel_based_approximate_compare_map_filter/node.cpp:128: [2] (buffer) memcpy:			destination		
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_compare_map_segmentation/src/voxel_based_compare_map_filter/node.cpp:83: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_compare_map_segmentation/src/voxel_based_compare_map_filter/node.cpp:84: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_compare_map_segmentation/src/voxel_based_compare_map_filter/node.cpp:85: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_compare_map_segmentation/src/voxel_based_compare_map_filter/node.cpp:89: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_compare_map_segmentation/src/voxel_distance_based_compare_map_filter/node.cpp:151: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_compare_map_segment	2	(buffer) memcpy:	Does not check for buffer overflows when copying to	(CWE-120).	Make sure destination can always hold the source data.

ation/src/voxel_distance_based_compare_map_filter/node.cpp:152: [2] (buffer) memcpy:			destination		
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_compare_map_segmentation/src/voxel_distance_based_compare_map_filter/node.cpp:153: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_compare_map_segmentation/src/voxel_distance_based_compare_map_filter/node.cpp:157: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_crosswalk_traffic_light_estimator/src/node.cpp:453: [2] (integer) atoi:	2	(buffer) memcpy:	Unless checked, the resulting number can exceed the expected range	(CWE-190).	if source untrusted, check both minimum and maximum, even if the input had no minus sign (large numbers can roll over into negative number; consider saving to an unsigned value if that is intended).
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_euclidean_cluster/lib/voxel_grid_based_euclidean_cluster.cpp:123: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120)	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_euclidean_cluster/test/test_voxel_grid_based_euclidean_cluster.cpp:68: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.

/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_ground_segmentation/src/ransac_ground_filter/node.cpp:310: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_ground_segmentation/src/ray_ground_filter/node.cpp:312: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_ground_segmentation/src/ray_ground_filter/node.cpp:317: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_ground_segmentation/src/scan_ground_filter/node.cpp:622: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_ground_segmentation/test/test_ransac_ground_filter.cpp:123: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_ground_segmentation/test/test_ransac_ground_filter.cpp:125: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_ground_segmentation/test/test_ransac_ground_fil	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.

ter.cpp:127: [2] (buffer) memcpy:					
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_ground_segmentation/test/test_ransac_ground_filter.cpp:129: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_ground_segmentation/test/test_ray_ground_filter.cpp:91: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_ground_segmentation/test/test_ray_ground_filter.cpp:93: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_ground_segmentation/test/test_ray_ground_filter.cpp:95: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_ground_segmentation/test/test_ray_ground_filter.cpp:97: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_ground_segmentation/test/test_scan_ground_filter.cpp:61: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_ground_segmentation/te	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.

st/test_scan_ground_filter.cpp:63: [2] (buffer) memcpy:					
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_ground_segmentation/test/test_scan_ground_filter.cpp:65: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_ground_segmentation/test/test_scan_ground_filter.cpp:67: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_image_projection_based_fusion/include/autoware/image_projection_based_fusion/segmentation_pointcloud_fusion/node.hpp:70: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_image_projection_based_fusion/src/roi_pointcloud_fusion/node.cpp:161: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filter_node.cpp:130: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filte	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.

r_node.cpp:131: [2] (buffer) memcpy:					
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filter_node.cpp:149: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filter_node.cpp:152: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filter_node.cpp:177: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filter_node.cpp:179: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filter_node.cpp:185: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filte	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.

r_node.cpp:188: [2] (buffer) memcpy:					
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filter_node.cpp:209: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filter_node.cpp:214: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filter_node.cpp:294: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filter_node.cpp:296: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filter_node.cpp:301: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filte	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.

r_node.cpp:374: [2] (buffer) memcpy:					
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filter_node.cpp:461: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filter_node.cpp:480: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filter_node.cpp:481: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filter_node.cpp:486: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filter_node.cpp:491: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_occupancy_grid_map_outlier_filter/src/occupancy_grid_map_outlier_filte	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.

r_node.cpp:497: [2] (buffer) memcpy:					
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_probabilistic_occupancy_grid_map/include/autoware/probabilistic_occupancy_grid_map/cost_value/cost_value.hpp:79: [2] (buffer) char:	2	(buffer) char:	Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues	(CWE-119!/CWE-120).	Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_probabilistic_occupancy_grid_map/include/autoware/probabilistic_occupancy_grid_map/cost_value/cost_value.hpp:101: [2] (buffer) char:	2	(buffer) char:	Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues	(CWE-119!/CWE-120).	Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_tensorrt_classifier/src/tensorrt_classifier.cpp:282: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_tensorrt_yolox/src/tensorrt_yolox.cpp:477: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_tensorrt_yolox/src/tensorrt_yolox.cpp:627: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_tensorrt_yolox/src/tensorrt_yolox.cpp:741: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_tensorrt_yolox/src/tenso	2	(buffer) memcpy:	Does not check for buffer overflows when copying to	(CWE-120).	Make sure destination can always hold the source data.

rrt_yolox.cpp:1233: [2] (buffer) memcpy:			destination		
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_tensorrt_yolox/src/tensorrt_yolox_node.cpp:199: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_tensorrt_yolox/src/tensorrt_yolox_node.cpp:199: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/autoware_tensorrt_yolox/src/tensorrt_yolox_node.cpp:200: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/perception_utils/src/run_length_encoder.cpp:47: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/perception/perception_utils/src/run_length_encoder.cpp:48: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/planning/autoware_freespace_planning_algorithms/src/rrtstar_core.cpp: 326: [2] (misc) open:	2	(misc) open:	Check when opening files - can an attacker redirect it (via symlinks), force the opening of special file type (e.g., device files), move things around to create a race condition, control its ancestors, or change its contents?	(CWE-362).	
/home/arthur/autoware/src/universe/autoware.universe/planning/autoware_f	2	(misc) open:	Check when opening files - can an attacker redirect it	(CWE-362).	

reespace_planning_algorithms/test/src/test_freespace_planning_algorithms.cpp:354: [2] (misc) open:			(via symlinks), force the opening of special file type (e.g., device files), move things around to create a race condition, control its ancestors, or change its contents?		
/home/arthur/autoware/src/universe/autoware.universe/planning/autoware_static_centerline_generator/src/centerline_source/bag_ego_trajectory_based_centerline.cpp:34: [2] (misc) open:	2	(misc) open:	Check when opening files - can an attacker redirect it (via symlinks), force the opening of special file type (e.g., device files), move things around to create a race condition, control its ancestors, or change its contents?	(CWE-362).	
/home/arthur/autoware/src/universe/autoware.universe/planning/autoware_static_centerline_generator/src/static_centerline_generator_node.cpp:406: [2] (misc) open:	2	(misc) open:	Check when opening files - can an attacker redirect it (via symlinks), force the opening of special file type (e.g., device files), move things around to create a race condition, control its ancestors, or change its contents?	(CWE-362).	
/home/arthur/autoware/src/universe/autoware.universe/planning/behavior_path_planner/autoware_behavior_path_lane_change_module/src/utils/utils.cpp:1099: [2] (integer) atoi:	2	(integer) atoi:	Unless checked, the resulting number can exceed the expected range	(CWE-190).	If source untrusted, check both minimum and maximum, even if the input had no minus sign (large numbers can roll over into negative number; consider saving to an

					unsigned value if that is intended).
/home/arthur/autoware/src/universe/autoware.universe/planning/behavior_path_planner/autoware_behavior_path_planner_common/src/utils/drivable_area_expansion/static_drivable_area.cpp:1200: [2] (integer) atoi:	2	(integer) atoi:	Unless checked, the resulting number can exceed the expected range	(CWE-190).	If source untrusted, check both minimum and maximum, even if the input had no minus sign (large numbers can roll over into negative number; consider saving to an unsigned value if that is intended).
/home/arthur/autoware/src/universe/autoware.universe/planning/behavior_path_planner/autoware_behavior_path_start_planner_module/src/start_planner_module.cpp:1854: [2] (buffer) char:	2	(buffer) char:	Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues	(CWE-119!/CWE-120).	Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.
/home/arthur/autoware/src/universe/autoware.universe/planning/behavior_path_planner/autoware_behavior_path_static_obstacle_avoidance_module/src/utils.cpp:260: [2] (integer) atoi:	2	(integer) atoi:	Unless checked, the resulting number can exceed the expected range	(CWE-190).	If source untrusted, check both minimum and maximum, even if the input had no minus sign (large numbers can roll over into negative number; consider saving to an unsigned value if that is intended).
/home/arthur/autoware/src/universe/autoware.universe/planning/behavior_velocity_planner/autoware_behavior_velocity_intersection_module/src/util.cpp:346: [2] (integer) atoi:	2	(integer) atoi:	Unless checked, the resulting number can exceed the expected range	(CWE-190).	If source untrusted, check both minimum and maximum, even if the input had no minus sign (large numbers can roll over into negative number; consider

					saving to an unsigned value if that is intended).
/home/arthur/autoware/src/universe/autoware.universe/sensing/autoware_pcl_extensions/include/autoware/pcl_extensions/voxel_grid_nearest_centroid_impl.hpp:177: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/sensing/autoware_pcl_extensions/include/autoware/pcl_extensions/voxel_grid_nearest_centroid_impl.hpp:220: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/sensing/autoware_pcl_extensions/include/autoware/pcl_extensions/voxel_grid_nearest_centroid_impl.hpp:271: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/sensing/autoware_pointcloud_preprocessor/include/autoware/pointcloud_preprocessor/passthrough_filter/passthrough_uint16.hpp:432: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/sensing/autoware_pointcloud_preprocessor/src/crop_box_filter/crop_box_filter_nodelet.cpp:140: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/sensing/autoware_pointcloud_preprocessor/src/crop_box_filter/crop_	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.

box_filter_nodelet.cpp:141: [2] (buffer) memcpy:					
/home/arthur/autoware/src/universe/autoware.universe/sensing/autoware_pointcloud_preprocessor/src/crop_box_filter/crop_box_filter_nodelet.cpp:142: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/sensing/autoware_pointcloud_preprocessor/src/crop_box_filter/crop_box_filter_nodelet.cpp:158: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/sensing/autoware_pointcloud_preprocessor/src/crop_box_filter/crop_box_filter_nodelet.cpp:161: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/sensing/autoware_pointcloud_preprocessor/src/crop_box_filter/crop_box_filter_nodelet.cpp:162: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/sensing/autoware_pointcloud_preprocessor/src/crop_box_filter/crop_box_filter_nodelet.cpp:163: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/sensing/autoware_pointcloud_preprocessor/src/downsample_filter/pickup_based_voxel_grid_downsample_filter_node.cpp:124: [2] (buffer) memcpy:	2	(buffer) memcpy:	Does not check for buffer overflows when copying to destination	(CWE-120).	Make sure destination can always hold the source data.
/home/arthur/autoware/src/universe/autoware.univ	2	(buffer) memcpy:	Does not check for buffer overflows	(CWE-120).	Make sure destination can

erse/sensing/autoware_pointcloud_preprocessor/src/downsample_filter/pickup_based_voxel_grid_downsample_filter_node.cpp:127: [2] (buffer) memcpy:			when copying to destination		always hold the source data.
/home/arthur/autoware/src/universe/autoware.universe/sensing/autoware_pointcloud_preprocessor/src/downsample_filter/robin_hood.h:1599: [1] (buffer) equal:	1	(buffer) equal:	Function does not check the second iterator for over-read conditions	(CWE-126)	This function is often discouraged by most C++ coding standards in favor of its safer alternatives provided since C++14. Consider using a form of this function that checks the second iterator before potentially overflowing it.
/home/arthur/autoware/src/universe/autoware.universe/sensing/autoware_pointcloud_preprocessor/src/downsample_filter/robin_hood.h:1600: [1] (buffer) equal:	1	(buffer) equal:	Function does not check the second iterator for over-read conditions	(CWE-126)	This function is often discouraged by most C++ coding standards in favor of its safer alternatives provided since C++14. Consider using a form of this function that checks the second iterator before potentially overflowing it.
/home/arthur/autoware/src/universe/autoware.universe/sensing/autoware_pointcloud_preprocessor/src/downsample_filter/robin_hood.h:1609: [1] (buffer) equal:	1	(buffer) equal:	Function does not check the second iterator for over-read conditions	(CWE-126)	This function is often discouraged by most C++ coding standards in favor of its safer alternatives provided since C++14. Consider using a form of this function that checks the second iterator before

					potentially overflowing it.
/home/arthur/autoware/src/universe/autoware.universe/sensing/autoware_pointcloud_preprocessor/src/downsample_filter/robin_hood.h:1619: [1] (buffer) equal:	1	(buffer) equal:	Function does not check the second iterator for over-read conditions .	(CWE-126)	This function is often discouraged by most C++ coding standards in favor of its safer alternatives provided since C++14. Consider using a form of this function that checks the second iterator before potentially overflowing it.