A Testing Framework for Evaluating Behavioural Correctness in Robotic Systems Using Clustering as an Oracle

by

MD Ishmam Labib Chowdhury

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
June 2025

Abstract

The integration of robotic systems is rapidly expanding across a wide range of industries, but this growth is accompanied by an increase in security risks. The large attack surface of robotic and AI-driven autonomous systems makes them particularly attractive targets for malicious actors. Their inherent complexity and the wide array of entry points leave them vulnerable to various forms of attack. Fuzz testing, a technique widely employed in software security, has proven effective in identifying vulnerabilities by generating random inputs to uncover weaknesses. However, applying fuzz testing to robotic systems presents two significant challenges: the lack of a reliable oracle to evaluate test outcomes and the difficulty of detecting failures related to behavioral correctness, as traditional fuzzing focuses on memory safety and software bugs.

In this work, we propose a novel testing framework for robotic systems aimed at addressing these limitations. Our central research question is whether feature vectors representing a robotic system's communication graph and messages can enable clustering to serve as a reliable oracle for assessing behavioural correctness. The framework involves recording experiment data, modeling communication as ROS graphs, extracting feature vectors from graphs and message data, and performing clustering to identify patterns. Using agglomerative hierarchical clustering on the generated

ROS bags and system graphs, we evaluate the effectiveness of this approach in distinguishing between normal and malicious experiment runs.

The results demonstrate high accuracy, with clustering models identifying malicious experiments with over 92% accuracy. However, metrics such as Area Under the Curve (AUC) suggest significant rates of false positives, limiting the model's ability to reliably differentiate between true and false anomalies. Interestingly, clustering based solely on structural features yielded lower accuracy but improved AUC values, highlighting its potential for better differentiation between malicious and non-malicious behaviors. This framework contributes to advancing the behavioral analysis of robotic systems, offering a new perspective on leveraging graph-based clustering for robust security testing.

Acknowledgments

I would like to express my deepest gratitude to everyone who has supported me throughout the journey of completing this thesis.

First and foremost, I would like to thank my family and friends for their support, encouragement, and understanding.

To my labmates, thank you for creating an environment of collaboration and intellectual curiosity. Your feedback, thought-provoking discussions, and camaraderie made this journey not only productive but also enjoyable.

I would also like to extend my thanks to my supervisor, Dr. Thomas Dean. Your guidance, patience, and expertise have been instrumental in shaping this research. Your belief in me and constant motivation have been a source of strength and inspiration during this challenging yet rewarding process.

Statement of Originality

I certify that this thesis is my own work and has not been previously published or infringed upon any copyright. Although the ROS 2 ATC simulation was developed in collaboration with lab members, the intellectual contributions and content presented here are the result of my independent effort. All external ideas, techniques, or quotations are appropriately acknowledged and cited following the IEEE Citation Reference manual.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The integration of robotic systems is rapidly expanding across a broad spectrum of industries, including transportation, manufacturing, and military operations, while also becoming increasingly prevalent in everyday life, with applications ranging from robotic vacuum cleaners to semi-autonomous vehicles. These systems often function in mission-critical environments where their performance can directly influence the safety of individuals, raising the importance of ensuring their operational integrity. As robotic systems evolve, they become more complex, incorporating a variety of interdependent components such as physical hardware, software algorithms, machine learning models, and large-scale data processing [42]. The large attack surface makes them an attractive target for malicious actors. Autonomous robots and AI-driven systems are especially vulnerable due to the wide array of entry points available for attacks. These threats can manifest in the form of compromised hardware, manipulated algorithms, or corrupted data, all of which could lead to severe operational failures or breaches of security [42]. As a result, securing these systems requires a comprehensive approach that addresses the broad and multifaceted attack surface they present.

Fuzz testing is a widely employed technique in various software systems, including robotic systems, to enhance security by identifying vulnerabilities. Fuzz testing, or fuzzing, involves the brute-force generation of random inputs to a system with the goal of uncovering weaknesses or exploitable flaws [55]. This method is particularly effective in detecting common security vulnerabilities that could be exploited by malicious actors. By subjecting the system to unexpected or malformed inputs, fuzzing can identify flaws that traditional testing methods might overlook. Moreover, the effectiveness of fuzzing can be improved through the use of intelligent input generation techniques, such as grammar-based fuzzing, which tailors inputs to the specific syntax and structure of the system under test [55]. These advanced techniques allow for more targeted exploration of the system's vulnerabilities, increasing the likelihood of identifying critical security risks and preventing potential attacks [55].

1.1 Problem

Fuzzing robotic systems presents two important challenges. First, there is often a lack of a reliable oracle to determine whether a test has passed or failed, making it difficult to assess the outcomes of the fuzzing process [25]. Second, it is challenging to identify failures related to behavioural correctness, as fuzzing traditionally focuses on detecting memory safety issues and software bugs [32]. In robotic systems, however, the behaviour of the robot is of great importance. Misbehaviour, even in the absence of software crashes, can be critical as improper actions or decisions by the robot could lead to dangerous or unintended consequences. Thus, ensuring both the functional and behavioural correctness of robotic systems is crucial for their safe and reliable operation.

1.2 Objective

In this thesis, we propose a testing framework for robotic systems designed to address the limitations previously discussed. The framework utilizes clustering techniques to detect misbehaviour within robot navigation simulations in a ROS2 environment [39]. By leveraging key properties of the robotic system, such as the messages exchanged between components and the overall graph structure of these components, the framework seeks to identify behavioural anomalies. Our central research question is: **Can we construct feature vectors representing a robotic system such that clustering can serve as a reliable oracle to assess behavioural correctness?** Through this approach, we aim to explore whether clustering can effectively identify misbehaviours that go beyond traditional software bugs, ensuring a more comprehensive evaluation of robotic systems' performance.

1.3 Motivation

Between October 2018 and March 2019, two Boeing 737 Max 8 flights tragically crashed shortly after takeoff in Indonesia and Ethiopia, resulting in the deaths of 189 and 157 passengers, respectively [45]. These incidents were linked to the introduction of a new flight control system, the Maneuvering Characteristics Augmentation System (MCAS), which was added to these aircraft without the pilots' knowledge [45]. The pilots struggled to counteract the MCAS, which repeatedly forced the aircraft's nose downward. In response, all Boeing 737 Max aircraft were grounded in March 2019 [45].

The function of the MCAS was to automatically lower the nose of the aircraft if the plane's sensors detect it is climbing too steeply, which could cause a stall

[22]. The MCAS system relies on data from an angle of attack (AOA) sensor, which measures the aircraft's angle relative to the oncoming airflow [22]. A malfunctioning sensor falsely indicated that the plane was at risk of stalling, causing the MCAS to repeatedly push the nose down, despite the fact that the aircraft was flying normally. The pilots were unable to correct this behaviour leading to the fatal crashes [22].

Although this incident was not caused by malicious actors, it prompts an important question about the system's ability to respond to deliberate malicious interference. How was a single point of failure introduced in a mission-critical system like a Boeing aircraft? A major contributing factor was insufficient testing. In software testing, the primary focus is often on identifying bugs. However, for mission-critical systems, it is equally important to test for behavioural scenarios and ensure correctness. Even if all components are free of software bugs, it is essential to verify that the system continues to function correctly when one component is compromised. The aim of this thesis is to design a testing framework that mitigates such risks by emphasizing behavioural correctness during the testing phase.

1.4 Contributions

This thesis presents a testing framework for assessing the behavioral correctness of robotic systems by analyzing communication graphs and message contents, independent of the system's underlying implementation. Our approach focuses on structural and data-flow characteristics to detect misbehaviour.

Additionally, we introduce a novel plugin-based architecture for modular feature extraction, enabling easy integration and removal of plugins. This work also facilitates the creation of datasets from malicious and non-malicious experiments, providing a

foundation for evaluating security risks in robotic systems.

1.5 Organization of Thesis

We begin by introducing the necessary background material in Chapter 2 and related works in Chapter 3, which are essential for understanding this thesis. Chapter 4 presents the testing framework, outlining the high-level steps required for its implementation. In Chapter 5, we detail the experimental setup within the ROS2 environment, describe the procedures used to conduct the experiments, and explain the implementation of the testing framework including our approach to clustering experiment runs. Chapter 6 discusses the clustering results and explores potential future improvements. Finally, Chapter 7 provides concluding remarks and suggests directions for future work.

Chapter 2

Background

In this chapter, we will provide the necessary background information relevant to this research, including an overview of ROS1 and ROS2, fuzzing techniques, and the oracle problem. We will also discuss the current gaps in this field, particularly the lack of an automated test oracle for classifying test inputs in real-time distributed systems.

2.1 Robot Operating System

The Robot Operating System (ROS) is a popular open-source framework used for developing and testing robotic applications. It is widely used in academia and prototyping, and serves as a great real-time distributed system for experiments.

ROS 1, the first iteration of ROS, was designed to be peer-to-peer, tools-based, multi-lingual, thin, and free and open-source. It consists of four fundamental concepts: nodes, messages, topics, and services[46].

In ROS, nodes are independent software modules that perform computations, capture data, and communicate with other nodes through messages. These messages contain

strictly typed data structures and are exchanged between nodes using the publisher-subscriber messaging protocol. Publishers send messages to topics, and subscribers read these messages by subscribing to the same topics. Topics can have multiple concurrent publishers and subscribers, and nodes can publish to or subscribe to multiple topics [46].

However, the topic-based publish-subscribe model is not suitable for synchronous transactions. ROS uses a service for this purpose instead. A service is defined by a name and a request and response message, which are both strictly typed. This is similar to how web services are defined by a URI and have specific request and response formats [46].

As the popularity of ROS 1 grew, research into its security revealed that it was inadequate [15]. In response, ROS 2 was introduced with a more secure architecture that includes the use of the Data Distribution Service (DDS) [20], improving the security infrastructure of ROS 2 compared to its predecessor.

ROS2 was designed with an emphasis on security, scalability, and maintainability [39]. It was built on top of the Data Distribution Service (DDS) middleware, which provides improvements in security for critical infrastructure [39]. In addition to the fundamental concepts of ROS, ROS2 also introduces actions, which are goal-oriented and asynchronous communication interfaces that include a request, response, periodic feedback, and the ability to be canceled. Actions are useful for long-running tasks, such as navigation in autonomous vehicles [39].

The ROS2 middleware adds layers of abstraction behind the ROS client library (rcl) to create a system-agnostic framework. This allows developers to choose from

different ROS middleware (rmw) frameworks provided by DDS vendors with minimal code changes [39]. The network interfaces in ROS2 (topics, services, actions) are defined using Message Types through an Interface Description Language (IDL) [39]. This helps to create a flexible and extensible system that can be used across a variety of platforms. DDS also introduces Quality-of-Service (QoS) parameters, which optimizes data delivery under unreliable settings based on the available bandwidth and latency, resolving issues seen in the TCP/IP implementation of ROS [39].

With the adoption of ROS2 and its increased usage in research and robotic applications, the security of the framework continued to be evaluated. Static code analysis on the ROS2 source code was done previously, where they discovered empty control flow statements and security configuration flags set to 'False' by default [7]. Encryption methods used in ROS2, such as MD-5 encrypted packets, were found to be susceptible to cryptanalysis, compromising message integrity [16]. In addition, hardware used in robotic systems can be attacked through side channel attacks [16]. The use of signed packets and message encryption also resulted in a performance tradeoff, with an increase in message latency of 113% [16]. Other research has also observed decreased performance in real-time robotic systems stronger security configurations in ROS2 [48]. Furthermore, ROS2 is not suitable for handling large message packets, which can pose a risk of DoS attacks [7, 41].

The open source nature of the ROS framework allows several useful additions, including ROS bags [34]. Bags are a persistent data recording mechanism which provide the ability to record and replay messages from a ROS system, stored in an SQLite database [34]. ROS bags are crucial for data analysis, logging and debugging. Each bagfile consists of a collection of recorded messages from every topic provided

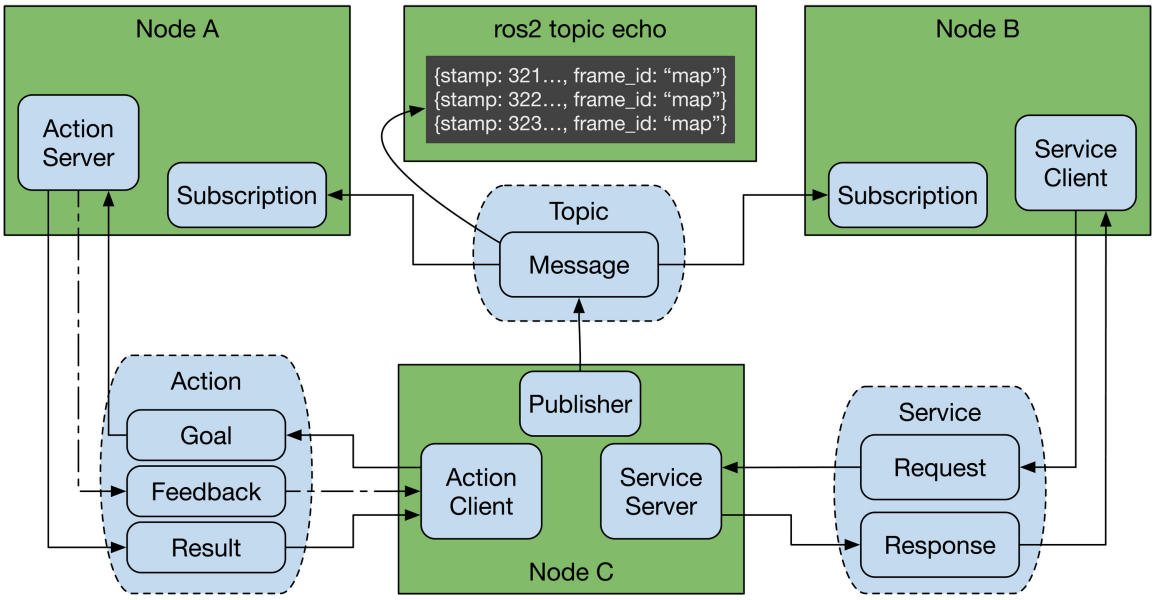


Figure 2.1: ROS2 Architecture Example [38]

when recording the bag. Each message in the bagfile contains a message ID, topic name, topic type, serialized message and the timestamp of the message; the serialized message can be deserialized using the topic type [34]. Each bag also contains metadata of the recordings; the list of all topics, message count and time range [34]. ROS bags can also be replayed with certain specifications using ROS2 services, including playing from a specific timestamp, setting the rate of playback, pausing and resuming playback, and replaying messages from specific topics [34].

Nav2 represents a sophisticated software framework created on the ROS environment, designed to simulate various navigation environments [37]. Our research leverages the framework to develop a testbed for autonomous navigation simulation. We harness Nav2's robust architecture to construct multiple controlled experimental settings. Through the application of fuzzing techniques, we analyse and evaluate the

behavioural impact on the autonomous capability of the Nav2 system. The modularity of Nav2 allows us to tailor simulations to suit our appropriate use case and accurately assess the impact on the system as a whole. We employ the collision monitor node built in to Nav2 to add an extra dimension to observe the behaviour of the autonomous vehicle. The collision monitor node in Nav2 actively monitors the robot's surroundings in real-time, detecting potential collisions with obstacles and dynamically adjusting the robot's trajectory to avoid accidents [37]. This is done using two polygons which either force the robot to slowdown or stop based on the proximity to an obstacle [37]; we observe the behaviour by modifying the data the collision monitor receives through these polygons.

2.2 Fuzzing

With the apparent security vulnerabilities of ROS 2, a robust and holistic testing framework is required to ensure a ROS application is secure. A heavily researched field of testing which can be used in ROS to find vulnerabilities and misbehavior is fuzz testing. Fuzz testing, or fuzzing, is a field in cybersecurity which revolves around finding overlooked vulnerabilities through an automated, brute forced process. Fuzzing involves randomly generating input and passing them through a system-under-test to find overlooked vulnerabilities which are often missed during the manual testing phase [49]. Many common malicious attacks can be prevented through this technique, including buffer overflow attacks, SQL injections, and cross-site scripting attacks [49].

Randomly generating inputs for a system-under-test is a robust and straightforward technique to find vulnerabilities; however, for a system with complex input

structures, this process can be inefficient and difficult. By modeling a grammar for the system's input, further inputs can be created efficiently and systemically, which can generate meaningful tests that properly test the system [55]. This is especially useful for a system with complex input types such as a ROS application.

Fuzz testing has been used to test ROS for vulnerabilities. One common area includes fuzz testing the DDS layer that ROS 2 is built on. Maggi et. al [40]. from Trend Micro Research applied fuzz testing to the 3 open source DDS implementations and found 6 new vulnerabilities which they deemed critical, including network amplification and reflection, buffer overflow allowing read/write access to memory stack/heap, and XML parsing vulnerabilities [40]. Liang et. al. applied fuzz testing to the ROS Middleware Interface (RMW) where they identified 9 previously unknown vulnerabilities including denial of service flaws and system crashes [35].

While fuzz testing is helpful for finding overseen vulnerabilities, a limitation lies in the lack of a reliable means to evaluate a passing or failing test, referred to as the oracle problem [25]. Test oracles can be placed in the following groups [25]:

- Explicit Oracles: These are contracts explicitly specified by the practitioner which define specific requirements.
- Implicit Oracles: These describe the general valid state of the program and are commonly used in general fuzzers in the form of sanitizers (memory, address, thread, floating-point, and undefined behavior sanitizers).
- Differential Oracles: Also known as pseudo oracles, they compare the behavior of different programs implementing the same specification.
- Metamorphic Oracles: These rely on metamorphic relations between inputs,

often used for cross-checking redundant operations in software APIs to construct equivalent API call test sequences and assertions.

- Dynamic Invariants: This approach monitors invariants between program variables during execution to identify unusual executions.

Furthermore, fuzzing can be used to determine whether an input crashes a system or not but to determine whether an input leads to malicious behaviour without crashing the system is an important challenge.

The most common solutions to the oracle problem involves mutation testing, metamorphic testing and cross referencing [25]. An alternative approach, using data science techniques to automatically cluster and label tests leads to less manual work [8]. We can assume failing test cases will fall in smaller, sparse clusters while passing test cases lie in large, dense clusters; using this assumption to cluster the behaviour of a system-under-test can assist in creating a testing framework which can identify behavioural anomalies.

The background presented in this chapter establishes the concepts and challenges addressed in this research. ROS, in its iterations of ROS 1 and ROS 2, provides a robust framework for real-time distributed systems, yet it exhibits vulnerabilities that necessitate rigorous testing frameworks [46, 39]. The evolution from ROS 1 to ROS 2 highlights the advancements in security, scalability, and modularity. Nav2, as an application within ROS 2, offers a platform for simulating autonomous navigation [37].

Fuzzing is an approach to uncover hidden vulnerabilities within such systems. While effective at identifying syntactic and memory flaws, it faces limitations, particularly the oracle problem, which challenges the ability to assess the correctness

of test outcomes reliably [25]. Existing solutions, such as metamorphic and mutation testing, provide partial remedies but often require significant manual effort [25]. The potential of clustering techniques to automate the identification of behavioral anomalies introduces a new approach to addressing these gaps [8].

This chapter underscores the importance of developing a comprehensive and automated testing framework to enhance the security and reliability of ROS-based systems. By leveraging the insights and methodologies outlined here, the following chapters will outline related works to these problems and detail a proposed approach to tackle these challenges.

Chapter 3

Related Work

3.1 Limitations of Fuzzing

Fuzzing is a widely used tool to find vulnerabilities in software through an automated, brute-force approach. We are able to improve this process by clearly defining the input grammar for a system and randomly generating structured inputs to discover meaningful vulnerabilities. However, even with a well structured input, vulnerabilities found by fuzzing are often limited to system crashes rather than misbehaviour.

There are three main fuzzing techniques, each with distinct benefits and drawbacks.

- **White box:** White box fuzzing describes fuzzing with input generation when the source code and input specification is available [25]. This allows for precise input generation used to target the system-under-test accurately. White box fuzzing was first introduced through the Scalable Automated Guided Execution (SAGE) fuzzer [24]; SAGE generates well-formed input based on the system's source code and symbolically executes the program to discover constraints from conditional branches to cover execution paths which have not been

tested [24]. SAGE ensures comprehensive test coverage for intricate systems with well-formed inputs, however, accessing source code remains uncommon for many programs.

- **Grey box:** Grey box fuzzing leverages instrumentation of a program to generate feedback-driven input [25]. Grey box fuzzing incorporates elements from both white box and black box fuzzing as the tester has partial knowledge of the internal workings of the software being tested. This knowledge can include information about the structure of the program, such as its APIs or certain algorithms, but not necessarily access to the complete source code. American Fuzzy Lop (AFL) uses an instrumentation-guided genetic algorithm to perform grey box fuzzing [2]. AFL uses code coverage feedback to guide the fuzzing process by instrumenting the program binaries to track which parts of the code are executed during testing. This feedback allows AFL to prioritize inputs that lead to new code paths, effectively guiding the fuzzer towards unexplored areas of the program [2]. Grey box fuzzing provides wide test coverage akin to black box fuzzing and targeted input generation seen in white box fuzzing but the efficacy of grey box fuzzing relies on partial knowledge of the system which cannot always be guaranteed. Feedback-driven input generation also introduces additional complexity to the fuzzer.

- **Black box:** Black box fuzzers operate under the assumption that the source code of the system-under-test is inaccessible, and instrumentation of the code is not feasible [25]. Essentially, the program is treated as a complete black box. This fuzzing method involves generating random test inputs to observe how the program behaves, particularly noting instances where it breaks down.

Although black box fuzzing is straightforward in its approach, its effectiveness is often limited; it tends to uncover only shallow bugs, and detecting abnormal behavior can prove challenging. Due to its simplicity and ability to provide broad test coverage, black box fuzzing remains prevalent in both industry and research settings. Notable examples of black box fuzzing tools include the Peach Protocol Fuzzer [21] and Radamsa [29].

Gopinath et al. underscore the validation of software behavior and the enhancement of test oracle quality as challenges in fuzzing [25]. They advocate for evaluating the effectiveness of a test oracle and fuzzer quality through mutation analysis [25].

Kim, in their work [32], notes that conventional fuzzing methods often overlook semantic correctness bugs, focusing predominantly on memory safety issues [32]. They propose RoboFuzz, a feedback-driven fuzzing framework, which facilitates the detection of correctness bugs by employing an "oracle handler" to scrutinize execution states against predefined correctness criteria [32].

3.2 Mutation Analysis

Gopinath et. al. analyze the application of mutation analysis to measure fuzzer and test oracle quality [25]. Mutation analysis can be used to generate a test suite by mutating test cases that cause defects in order to generate additional tests [43]. There has also been a shift in research to leverage mutation analysis in assessing the quality of a test suite by introducing artificial defects to the program and evaluating the performance of the test suite on the mutated program [43]. The efficacy of a test oracle can be evaluated using the mutation score metric which is the ratio of killed mutants to the total number of mutants introduced, however, [25] argue mutation

score can also be used to assess fuzzer quality. Fuzzers are often evaluated on either code coverage or industry benchmarks with seeded bugs [36] [10] [17]. Gopinath et. al. argue that fuzzers should also be evaluated by discovering bugs that cause various types of incorrect behaviour, not solely program crashes [25].

By using mutation analysis to exhaustively seed a program with first order variations, a wide range of unbiased mutations are created which represent a diverse set of potential issues that could arise in the software. By doing so, mutation analysis prevents fuzzers from becoming too specialized in specific bug categories, providing a more comprehensive assessment of software vulnerability and avoids overfitting of the fuzzer [25].

The main obstacle encountered in mutation analysis lies in its substantial computational demands. Testing each mutation of the program against every input in the test suite increases the computational requirements, particularly with large, complex programs with many first order mutations. However, ongoing research in this domain shows promising strides toward addressing this challenge [33] [26].

3.3 RoboFuzz

The realm of robotic systems presents unique challenges for developing fuzzers and test oracles, as highlighted by Kim et al. These challenges include the diverse nature of robotic systems, their vast state spaces, and the presence of noisy hardware [32]. To address these challenges, Kim et al. devised the RoboFuzz framework to tackle these obstacles, along with the specific challenge of detecting semantic correctness bugs rather than just memory safety issues.

The RoboFuzz framework views a robotic system built on ROS as a collection of

distributed processes that exchange data. By capturing the data flow among these nodes, the fuzzer can evaluate the robot's behavior, effectively addressing the issue of heterogeneity [32]. The framework's components can be summarized as follows:

1. System Inspector: Creates ROS network graph (topological graph of system consisting of all nodes, topics, and message types [46])

2. Message Mutator: Generates and transfers type aware messages to stress ROS nodes; mutation method is inherited from AFL [2] and applied on ROS message types [32]

3. Hybrid Executor: Runs the robot in real world and simulated environments simultaneously and assesses the behaviour with mutated messages in both scenarios [32]

4. Oracle Handler: RoboFuzz provides an oracle template to allow user-defined oracles to assess the fuzzer and identify correctness bugs [32]

5. Feedback Engine: The authors state that coverage guided feedback is less effective in robotic systems as the behaviour is driven by data flow between nodes rather and so have a limited number of code flows [32]; they propose the idea of semantic feedback which uses domain knowledge of robotic systems to identify bugs [32]. Examples of this feedback include redundant sensor inconsistency, control error-based feedback, and cyber physical discrepancy feedback [32].

. During its evaluation of the $PX4$ drone system, the RoboFuzz framework successfully uncovered 30 previously unidentified bugs within the ROS environment. While RoboFuzz demonstrates its efficacy in testing ROS-based systems, creating oracles

demands a considerable level of domain expertise. Furthermore, RoboFuzz necessitates an independent model and primarily emphasizes general fuzzing rather than targeting malicious behavior.

3.4 Fuzzing in Robotic Systems

The application of fuzz testing within the ROS framework has provided strong results. RoboFuzz presents a framework that detects semantic correctness bugs through an innovative approach, incorporating coverage-guided feedback, analysis of the ROS network graph, and an oracle handler to verify correct robot behavior [32]. The ROFER framework enhances ROS fuzzing by introducing two novel techniques [9]:

1. multi dimensional input mutation which considers the impact of each input dimension to generate efficient test cases

2. message guided feedback, rather than coverage guided feedback, to reflect state transitions of the robot

Ella Duffy, fellow MSc. student and lab mate, proposes a domain-specific language applying fuzzing techniques to generate a malicious ROS node which tricks the system into performing abnormal behaviour [18]. The threat model for Ella's work revolves around the scenario where a sensor in the ROS system is compromised to send malicious, inconsistent data with the uncompromised sensor leading to anomalous behaviour [18]. The diagram in 3.1 represents a simplified model of a ROS system to illustrate the approach to fuzzing input data. In reality, the ROSTarget and ROSActuator can be arbitrary complex ROS systems. The attack kill chain is comprised of the following components:

- **ROSReplay:** the raw input provided to the ROS system which can be the real world inputs or replaying a recorded bag
- **ROSFuzzSensor:** node which provides normal messages to ROSFuzzer to mutate and compromise system
- **ROSUnfuzzSensor:** all nodes in the system not providing messages to fuzzer
- **ROSFuzzer:** node acting as compromised sensor, which mutates input from the ROSFuzzSensor and outputs to the ROSTarget in order to change the behaviour of the system
- **ROSTarget:** the algorithms which compute specific functions for the ROSActuator using input from the rest of the nodes
- **ROSActuator:** the control of the ROS system; this component determines the behaviour of the robot such as movement, collision detection, and velocity

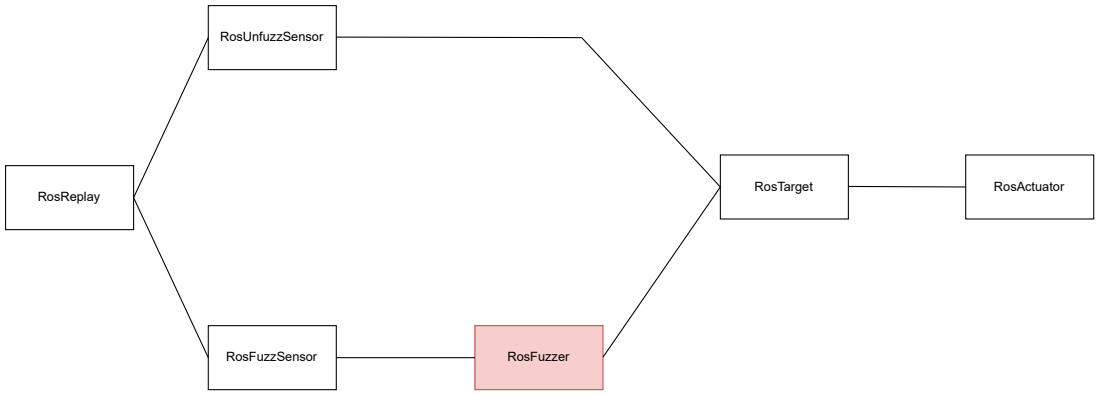


Figure 3.1: Diagram of ROS Attack Kill Chain [18]

The presented attack was carried out in two scenarios, an Air Traffic Control

(ATC) system simulated on ROS2. In the ATC system, the sensors consist of ground-based primary and secondary surveillance radars, as well as Automatic Dependent Surveillance-Broadcast (ADS-B) sensors. The ADS-B system enables aircraft to transmit their speed, position, altitude, and direction to cost-effective ground stations [6]. The ADS-B sensors are the compromised sensors in these experiments. The ROSTarget in this system is the ATC Fusion node, which is tasked with integrating data from all sensors to provide a unified and coherent representation of the airspace. The ROSActuator represents the collision detection algorithms and the control interface stations, which play a crucial role in ensuring safe airspace management.

The fuzzer will subscribe to the ADS-B sensor topic and feed compromised messages to the Fusion node. The Fusion node relays data to the actuators which include collision detection and air traffic controller [18].

The first set of scenarios tests modifying the sensor message to appear as if an aircraft has been deleted based on conditions; deleting on specific plane identification number, based on timer and plane callsign, and creating a deadzone which deletes all aircrafts when they are near a given location. The second set of scenarios creates fake data, i.e. a "ghost plane", which is observed by the ATC visualizer system. The ghost plane was creating in both a static scenario where the threat is mitigated relatively easily as well as a ghost plane following an existing plane appearing as real data [18]. The last set of scenarios imitates a message modification attack by updating the data of existing planes in the ATC system. Experiments for this scenario include making it appear that the plane is decreasing altitude, altering GPS positions, and modifying the messages of a plane to make it follow the plane it is closest to [18].

In the Nav2 system, the attack attempts to misguide a robot navigating from point

A to point B by compromising a LiDAR and IR sensor node and modifying messages sent to the collision monitor algorithm [18]. A similar message modification attack was carried out where the LiDAR and IR sensor messages were modified to cause the robot to stop moving as it believes there is a collision. In the next experiment, the author applies a smart fuzzer to the LiDAR sensor to simulate an advanced fake object [18]. In this experiment, the original LiDAR sensor keeps it's integrity and observes the correct information while the fuzzed sensor has it's output modified where it appears as if a fake object is lying at a fixed position in the simulation. As a result, the robot is unable to complete the navigation task as a perceived obstacle lies in it's path, which represents a successful modification in robot behaviour [18].

These scenarios were created manually using Python code which requires significant domain knowledge. To make this testing approach generalized, the author, Ella Duffy, creates a Domain Specific Language (DSL) to auto generate test cases given simple input from the robotics developer [18]. There are three required elements and four additional elements for the DSL:

- *Types*: The ROS2 message types which need to be imported in Python
- *Main Topic*: The input and output topic for the fuzz operation
- *Transform*: The fuzzing procedure; how the behaviour of the system will be modified. There are two sections in *Transform*; *Always*, specifies Python code which should run everytime a message is received, and *Action*, which contains the condition (*Filter*) and code to run (*Code*) to perform the fuzz operation [18]
- *Extra Source*: describes an extra subscription separate from the main topic.

Each extra source requires the *Ros Type* and *Preprocessing* parameters.

- *Parameters*: allows fuzzer to retrieve values on start up. For example, we can use the *Parameters* element to define the x and y coordinates of the fake object on start up

- *Required Storage and Constants*: values which will be created as in the initializer of the fuzzer node. These include values needed for computation and to create class attributes to use in future fuzzing operations [18]

- *Functions*: Composed of *Function Name*, *Function Parameters*, and *Code* elements, *Function* improves readability and advanced functionality of the fuzzing system by creating helper functions which can be used by complex filters

3.5 Clustering Failures

As seen by the drawbacks of RoboFuzz, creating an effective oracle for a fuzzer requires considerable domain knowledge which limits the general applicability of many testing frameworks [32]. One approach to create a generalised oracle is by performing anomaly detection techniques on a system's input and output pairs. In their preliminary study of this method, Almaghairbe et. al. hypothesize that failures will group into smaller clusters across test runs [8].

Clustering is a method for anomaly detection which analyzes large datasets by grouping the data into compact, meaningful clusters that represent the entire dataset [50]. By applying clustering to a test oracle context, we can identify inputs/outputs which deviate from the expected pattern and may indicate faulty behavior [8]. This can be effective to find behavioural correctness bugs as well as memory safety bugs.

In the study done by Rafig Almaghairbe and Marc Roper, the authors apply the hierarchical agglomerative clustering algorithm [8]. Hierarchical agglomerative clustering is a bottom-up approach that starts with each data point as an individual cluster and repeatedly merges the closest pairs of clusters until all points form a single cluster. Conversely, hierarchical divisive clustering is a top-down approach that starts with all data points in one cluster and repeatedly splits them into smaller clusters until each point is in its own cluster [12].

Almaghairbe et. al. test the heirarchical agglomerative clustering based oracle on two Java systems; nanoXML, an XML parser with five version containing faults, and Siena (Scalable Internet Event Notification Architecture), an event notification middleware with seven faulty versions [30][53]. The experiments consisted of running both the original and faulty versions of each system using the provided inputs to generate outputs; the clustering algorithm was then applied to these input/output pairs [8]. The effectiveness of the oracle can be assessed since the failures and associated faults are known.

The results from the clustering algorithm show that failures tend to group together in small clusters in both the nanoXML and Siena tests, supporting the initial hypothesis [8]. Specifically, the authors observed that when the cluster sizes are 15% to 20% the number of test cases, over 60% of the data points are failures [8].

3.6 MalChain

As seen in the RoboFuzz testing framework, one method to generalizing the testing approach of a ROS system is to analyse the data exchange patterns within the system [32]. We also saw that applying clustering algorithms can create an effective test

oracle [8]. Malchain, an approach for profiling virtual applications, combines these two techniques [23].

Malchain operates in the domain of virtualized applications which use the microservices architecture, where each specific task (logging, storage, monitoring, etc) are containerized and run independently [23]. Malchain analyzes the behaviour by examining the data exchange patterns of the system calls in each microservice [23]. The methodology of the Malchain framework can be summarized as:

1. System calls from the OS layer originating from Docker containers are collected. These calls (eg. read, write, send, receive) are filtered to accurately represent the activities of the target virtual application [23].

2. A role ID is assigned to each microservice in order to handle the case where a container is taken down and brought up with a new process ID. This step is to overcome the challenge of arbitrary and temporary process IDs and associate each system call with the correct microservice [23].

3. A microservice data exchange graph (MSDEG) is constructed, which is a directed graph that maps data exchanges between virtual process role IDs and various virtual resources over a defined period using a sliding window approach. In this graph, the vertices represent virtual process roles, I/O entities, and memory sections, while the edges indicate the direction of the data exchanges [23].

4. Fixed sized feature vectors are extracted from the MSDEG to represent the behaviour of the system. The feature vectors consist of Entropy, Variance, and Flow Proportion as the local features and Closeness Centrality and Betweenness

Centrality as the global features [23].

5. The feature vectors extracted in step 4 are used to train a machine learning clustering model to detect abnormal behaviour. The algorithms used in the Malchain framework are Isolation Forests, OneClassSVM, and EllipticEnvelope algorithms [23].

The authors applied the Malchain framework on four reproduced attack scenarios: virtual IP Multimedia Subsystem (IMS) [23], Flag Variable Overwritten Attack (FVOA) [13], Regular expression Denial of Service (ReDoS) [14], and Directory Harvest Attack (DHA) [19]. The training of the anomaly detection model used 80% of normal behaviors, while the evaluation used 20% normal behavior and 100% malicious behaviour [23]. Performance metrics included false alarm rate (FAR) and detection rate (DR) where the goal was to achieve a high DR and low FAR. At 5% FAR, Isolation Forest had the best average detection rate, while at 1% FAR, One Class SVM performed best [23]. Overall, the Malchain framework applied to the given test scenarios was able to detect anomalous behaviour with a low false positive rate and high detection rate.

Conclusion

Fuzzing has proven to be an invaluable tool for software testing, uncovering vulnerabilities by generating and testing a wide array of inputs. However, traditional fuzzing techniques, including white-box, grey-box, and black-box approaches, face challenges in detecting behavioural correctness bugs and ensuring comprehensive test coverage [32]. This chapter has outlined the strengths and limitations of these methods, emphasizing the need for innovative approaches to address these shortcomings.

Emerging techniques such as mutation analysis, domain-specific frameworks like RoboFuzz, and clustering-based anomaly detection provide promising directions to overcome these limitations. Mutation analysis improves the robustness of test oracles and fuzzer quality, while RoboFuzz demonstrates success in robotic systems, particularly in identifying semantic correctness issues. Clustering failures, as exemplified in hierarchical agglomerative clustering, offer a generalized method for identifying anomalies in system behavior. Frameworks like MalChain further illustrate the potential of combining data exchange pattern analysis with machine learning to detect abnormal behaviors in virtualized environments.

Chapter 4

Testing Framework

The limitations of fuzzing in ROS systems have been clearly outlined in the previous chapter. Fuzz testing frameworks rarely identify bugs beyond software crashes and memory safety issues. Additionally, creating an appropriate oracle to distinguish between successful and failing tests requires significant domain knowledge, making it difficult to develop a general approach to fuzz testing in ROS systems.

In our work, we will present a solution to the oracle problem by applying clustering techniques to tests on a ROS system compromised using the fuzzing approach from Ella Duffy's framework. This solution consists of four key components:

1. Record Experiments
2. Build Graph to Model Communication
3. Extract Feature Vectors from Graphs and Messages
4. Cluster Vectors

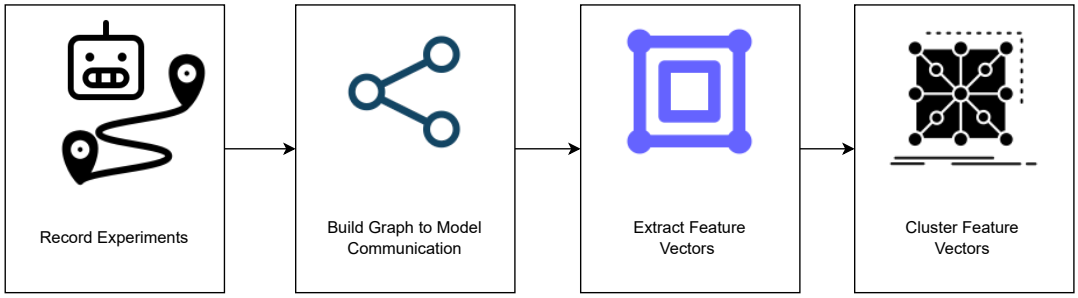


Figure 4.1: Testing Framework Overview

4.1 Record Experiments

The first component involves designing and documenting experiments to accurately depict scenarios where the ROS system can be compromised and attacked by a malicious actor. These experiments should include both normal operation and attack conditions. In this step, we use Ella Duffy's fuzzing node to compromise a LiDAR sensor in a Nav2 simulation. Additionally, we record each experiment run using ROS bags [34] to facilitate offline analysis and feature extraction, rather than during the experiment. By capturing the system's behavior in both normal and compromised states, we create a comprehensive dataset that highlights the differences between these states, enabling more effective detection of anomalies and potential attacks. This approach ensures that we have detailed and accurate records for subsequent analysis, enhancing the robustness of our solution.

4.2 Build Graph to Model Communication

Similar to the RoboFuzz framework [32] and the MalChain framework [23], we can view the ROS system as a distributed data exchange network. Building on this concept, we hypothesize that changes in the collection nodes and topics in a ROS system

can indicate potential malicious behavior. During our experiments, as we record the ROS bags, we create graphs that model the communication patterns between ROS nodes and topics over a period of time. Each 10 second interval of the experiment will consist of different features creating periodic graphs of the experiments. By analyzing these graphs, we aim to detect anomalies that may suggest compromised system behavior.

4.3 Extract Feature Vectors from Graphs and Messages

We created a plugin-based approach to generate feature vectors from the constructed graphs and messages replayed from the ROS bags. Drawing inspiration from the MalChain framework, we utilize features derived from graph properties to create vectors that characterize the communication system's structure [23]. To enhance this method, we also extract features from the messages within the ROS system based on the system's expected behavior. Although this process requires domain knowledge, our intuitive plugin-based architecture allows system owners to easily add or remove features from the feature vectors. This flexibility results in a more generalized and adaptable testing approach, accommodating various scenarios and system configurations. By enabling dynamic customization, our approach ensures that the testing framework can evolve alongside the ROS system, maintaining its relevance and effectiveness in detecting anomalies and potential malicious behaviors.

4.4 Cluster Vectors

Once the feature vectors are extracted from the ROS network graphs and messages, we apply clustering algorithms to all or a selected subset of the experiment runs. According to the findings of Almaghairbe et al. [8], tests that fail or exhibit anomalous behavior tend to form smaller, dense clusters. This clustering helps in identifying patterns indicative of faults or malicious activity within the ROS system. Furthermore, since the experiment designers have prior knowledge of which runs are malicious or non-malicious, we can validate the clustering results, effectively employing a semi-supervised approach. This validation step not only enhances the accuracy of our anomaly detection but also ensures that the clustering method is fine-tuned to distinguish between normal and compromised states effectively. By leveraging both unsupervised clustering techniques and supervised validation, we aim to create a robust and reliable framework for detecting and analyzing anomalies in ROS systems.

4.5 Experiment Types

We run this framework on a autonomous navigation system on the Nav2 library for ROS 2. The main task of the robot simulation will be to navigate from Point A to Point B using two LiDAR sensors and an Infrared (IR) sensor to identify obstacles and potential collisions, using the collision monitor node provided in Nav2 [37]. We compromise a sensor using Ella's fuzzing node to introduce a fake object in the path of the robot and prevent the robot from navigating to it's destination. There are six experiments which we run:

Experiment 1: A normal run with an additional node to act as a passthrough for the non compromised LiDAR sensor. We need the additional passthrough

node as fuzzing the sensor will introduce a delay in relaying messages to the target node. The passthrough will introduce a delay without modifying the messages, ensuring the timing of messages not being the sole feature in malicious behaviour detection.

Experiment 2: In a similar approach, this experiment is a normal run as well with an additional node subscribed to the odemetry topic, `/odom`. This is to ensure graph structure modifications do not result in false positive anomalies as the fuzzing node will introduce an additional subscription to `/odom`. We want to determine whether the graph structure change is the overriding signal or does other contextual information help distinguish malicious and non-malicious experiment runs.

Experiment 3: We introduce a node which adds random noise to the compromised LiDAR sensor to simulate a malicious actor attacking the system.

Experiment 4: To extend the attack, this experiment introduces the intelligent fuzzing node where the navigation system perceives a fake object exists in the simulation which can block the path of the robot

Experiment 5: Following experiment 2, we delay the start of the additional `/odom` subscription to identify whether changes in the system structure during the experiment run will affect the clustering algorithm during normal behaviour

Experiment 6: Similarly, we run the fake object node delayed to observe system changes mid experiment run during malicious behaviour

4.6 Running the Experiments

The high level run of each experiment can be broken down into the following steps:

1. Generate random starting coordinates of robot
2. Generate random target coordinates for navigation
3. Start base Nav2 simulation for autonomous navigation including the `collision_monitor` node
4. Start fuzzing node or pass through node depending on experiment type
5. Start `rosbag` recording for all topics and create ROS network graph
6. Send ROS `action` to start autonomous navigation
7. Terminate all nodes once robot reaches destination or timeout is reached

4.7 Results

Using the generated rosbags and ROS network graphs, we employ an agglomerative heirarchical clustering model to identify patterns in the experiments runs. The parameters for the clustering model were chosen by performing a grid search and selecting the combination of parameters that resulted in the best accuracy.

The clustering results overall showed a high accuracy in identifying malicious experiments - greater than 92% accurate on average - but other metrics such as Area-Under-Curve (AUC) suggested high rates of false positives. Clustering using only structural features resulted in a lower accuracy but a model which is better at differentiation malicious and non-malicious experiments.

Chapter 5

Experiments

5.1 Air Traffic Control System Setup

To familiarize ourselves with the ROS environment and messaging network, M.Sc. student Ella Duffy and I created an ATC (Air Traffic Control) system in ROS 2 in order to compromise the MCAS (Maneuvering Characteristics Augmentation System) sensor and send inconsistent data to the TCAS (Traffic Collision Avoidance System) sensor leading malicious behaviour [18]. In the setup for this experiment, I helped set up the fuzzing environment and created process to run and record experiments using ROS 2 bags. This system was created on the Ros 2 Foxy distribution [39].

To achieve this, we:

- Utilised an ATC system within the ROS 2 framework, focusing on the compromise ADS-B sensor and the Fusion node
- Established a fuzzing environment tailored to test the ATC system by introducing a node responsible to fuzz ADS-B data and provide it to the Fusion node

- Created procedures for running and recording experiments, utilizing ROS 2 bags to capture data from each run. This allowed us to analyze the effects of our manipulations on the system

5.2 Nav2 Setup

As discussed in Chapter 2, Nav2 is a software framework created on the ROS environment designed to simulate autonomous navigation [37] and is the test bed used to run all experiments. Additionally, the collision monitor node, used to monitor and detect collisions, is also used as a target node for the compromised sensor.

The setup for the experiments requires modifications in the Nav2 configurations, which is done directly on the config files used to start the nodes used by Nav2.

Changes to `nav2_bringup` config

The Nav2 framework provides multiple bring up systems to start simulations. Our experiments use the base bring up system provided, `nav2_bringup`, due to the offered flexibility and ability to modify several components [37].

The `nav2_bringup` offers multiple launch configurations; our experiments used the TurtleBot3 configuration with slight modifications which will be discussed further in the next section. To run the launch script, the following command was used:

```
ros2 launch nav2_bringup tb3_simulation_launch.py headless:=False
```

In this command, we are launching the `tb3_simulation_launch.py` node from the `nav2_bringup` package. The flag `headless:=False` indicates that the simulation should not run in headless mode, meaning it should display GUI elements (like

Gazebo's graphical interface) rather than running in a background-only mode [37].

This launch script orchestrates the setup and execution of a TurtleBot3 simulation environment using ROS 2. It begins by declaring various launch arguments for customization such as namespace, SLAM (Simultaneous Localization and Mapping) usage, map file, simulation time, and parameter files. The script prepares the simulation environment by generating a Simulation Description Format (SDF) file from an Xacro world model and spawns the TurtleBot3 robot with specified initial poses. SDF is an XML-based format used to describe robotic models and environments in robotic simulations including Gazebo [3]. The SDF is then modularized and parameterized using the Xacro extension of XML, which allows for macro substitution within XML files [1].

Finally, it launches essential nodes for robot state publishing, RViz (ROS Visualization), and the navigation stack , ensuring the functionality of the TurtleBot3 simulation.

In our experiment runs, we required the initial pose (coordinates) of the robot to be random on every run. There are two steps required to complete this requirement:

1. **Generate Random Pose**

   To create the random pose, we need to generate `x` and `y` coordinates that are both within the boundaries of the current simulation world and do not lie on any obstacles on the map. We find the boundaries of the map as well as obstacles by parsing the SDF.

   The SDF consists of keys for the borders and collisions; the borders on the map used in the simulation are `head, left_hand, right_hand, left_foot, right_foot, body` and the obstacles are `one_one, one_two, one_three, two_one,`

`two_two`, `two_three`, `three_one`, `three_two`, `three_three`.

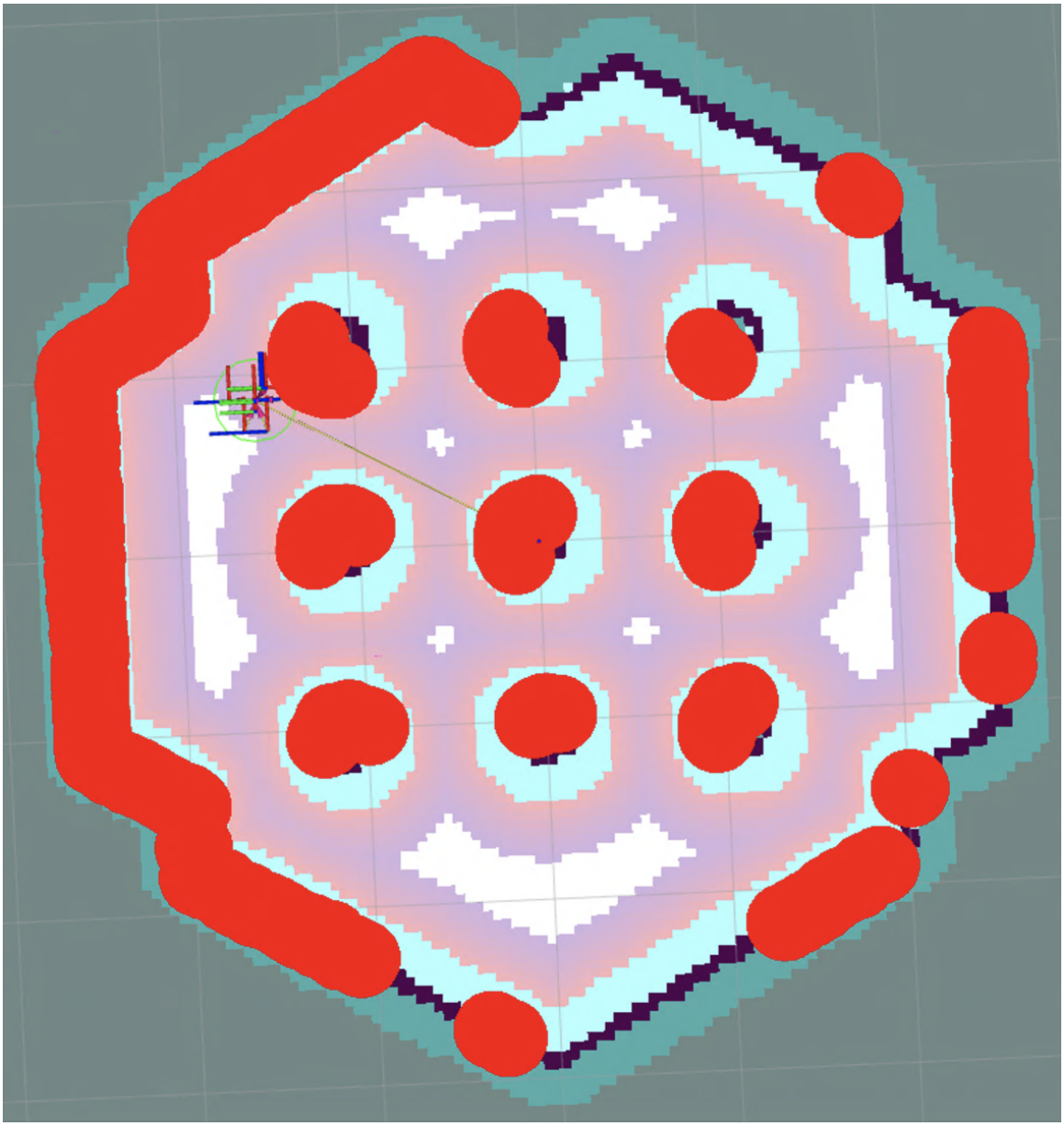


Figure 5.1: Experiment view in RViz. Consisting of boundaries, obstacles, and TurtleBot3

Then, we generate random `x` and `y` between the two border poses parsed from the SDF. We check whether the generated random pose lies on top of an obstacle.

If the pose is on an obstacle, the pose is regenerated; otherwise, it will be used as the initial pose of the simulation.

2. **Set New Initial Pose**

   The set the initial pose to the generated random pose, we have to require the parameter YAML (markup language often used for configurations) file used by `nav2_bringup`. This YAML file allows developers to specify settings that influence how a robot navigates its environment [37]. Parameters can include details such as sensor characteristics, dimensions of the robot, configurations for path planning algorithms, setup for obstacle avoidance mechanisms, parameters related to map management and localization techniques. By separating these parameters into a YAML format, Nav2 allows for modularity and ease of management without modifying the underlying codebase.

   The parameter YAML includes the AMCL (Adaptive Monte Carlo Localization) configuration which contains the `ros_parameter` attribute that we can modify to set the initial pose of the simulation. The `amcl` parameter is responsible for localizing the robot within a static map using the Adaptive Monte-Carlo Localizer (AMCL) [37]. AMCL estimates the pose of a robot in a given known map using a 2D laser scanner [28]. Within the `ros_parameter` in `amcl`, we can set `set_initial_pose` to `true` and add the generated pose here. The additional parameter will follow the format seen in Listing 5.1.

Changes to TurtleBot3

TurtleBot3 is a versatile robot platform built on ROS, tailored for educational, research, hobbyist, and prototyping applications [47]. TurtleBot3 distinguishes itself

Listing 5.1: Nav2 Bringup Config Changes

```
amcl:
    ...
    ros_parameters:
        ...
        set_initial_pose: true
        initial_pose:
            x: 0.0
            y: 0.0
            z: 0.0
```

with its compact size, robust capabilities in SLAM, navigation, and manipulation. It is designed to be highly customizable, allowing for various mechanical configurations and optional components like sensors and computers. This adaptability positions TurtleBot3 as an ideal test bed for research purposes [47]. TurtleBot3 is offered in two different modes, Burger and Waffle [47]. In the Nav2 simulation used to run our experiments, we used a TurtleBot3 Waffle simulated using Gazebo and RViz.

TurtleBot3 comes equipped with a single laser scanner but for our purposes, we require an additional sensor in order to add system redundancy and compromise the data sent to the target node. In Ella's work, she modified the configuration of the Turtlebot3 robot to include an additional LiDAR sensor as well as an IR sensor [18]. This was done by modifying the Unified Robot Description Format (URDF) and SDF files within the Nav2 simulation configuration. The URDF file is an XML representing the robot configuration consisting of links and joint motions, which is used by the Gazebo and RViz simulation software [5]. The final robot used in our experiments include three sensors:

- `base_scan`

  This is the original 360-degrees LiDAR sensor on the base TurtleBot3 robot.

This sensor publishes messages of type `sensor_msgs/LaserScan` to the topic `/scan` [37]

- `base_scan_tom`

  The first sensor added by Ella, an additional 360-degrees LiDAR sensor which publishes messages of type `sensor_msgs/LaserScan` to the `/scan_tom` topic [18]

- `base_ir`

  The second sensor added by Ella is an IR sensor on the top level of the robot which publishes `sensor_msgs/Range` messages to the topic `/range_ella` [18]

Changes to `collision_monitor` and `cost_map`

As discussed previously, our experiments involve using the `collision_monitor` node provided by the Nav2 framework as a target for our compromised node. As new sensors have been added to the robot, these modifications need to reflected in the configuration of the `collision_monitor` node [37]. Similar to `nav2_bringup`, the `collision_monitor` node can be changed directly on the parameter YAML file `collision_monitor_params.yaml`. The `collision_monitor` node configuration needs to be updated to include the two additional sensors as observation sources for the collision monitoring algorithm, which is added by including the topics where the sensors publish their respective messages to [18], seen in Listing 5.2.

With the introduction of the fuzz node, an additional topic was added to the collision monitor configuration YAML with the topic name `/scan_fuzz` [18]. The addition of the fuzz topics also require modifications to the `cost_map`, which refers

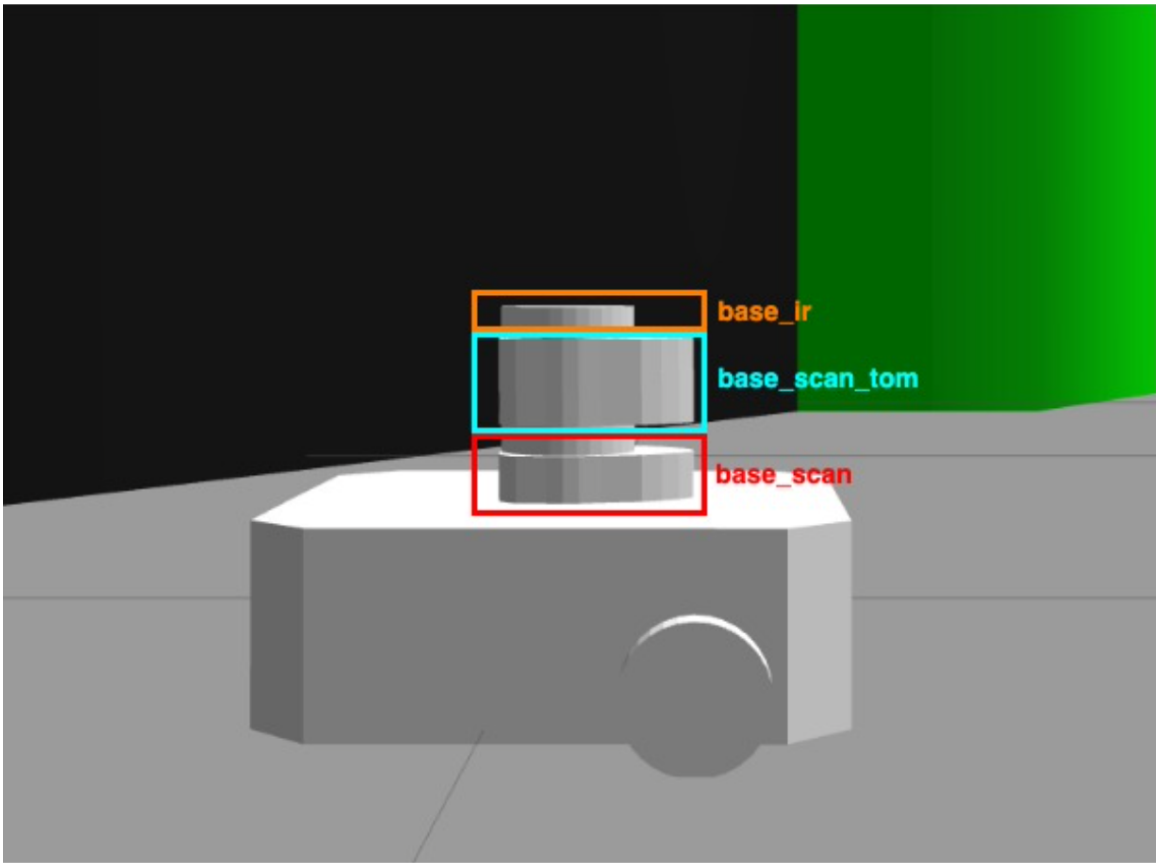


Figure 5.2: Turtlebot3 Breakdown [18]

to environmental representation grid that supplies a robot with crucial perception information for navigation [4]. There are two types of costmaps: the global costmap, which maintains data for overall navigation, and the local costmap, which maintains data for local navigation [4]. Plugins are added to the `cost_map` configuration in order to add the additional observation sources from the fuzzed topics [18].

This section outlined the setup of two primary experimental environments: the ATC system and Nav2, both implemented in ROS 2. The ATC system provided a platform to test sensor compromise methods in an air traffic context, utilizing ADS-B data fuzzing to induce anomalous behavior in the Traffic Collision Avoidance

Listing 5.2: Collision Monitor Config Changes

```
scan_tom:
    type: "scan"
    topic: "/scan_tom"
range_ella:
    type: "range"
    topic "/range_ella"
scan_fuzz:
    type: "scan"
    topic: "/scan_fuzz"
```

System (TCAS). The Nav2 framework served as a testbed for autonomous navigation experiments with TurtleBot3, necessitating extensive configuration modifications for sensor integration, random initial poses, and customized collision monitoring with a compromised sensor. These setups provide a method for executing experiments and capturing data in support of our research objectives.

5.3 Running the Experiments

To begin running the experiments, we need to start the Nav2 simulation using the `nav2_bringup` node as described previously:

```
run_nav2_bringup = f"ros2 launch nav2_bringup
tb3_simulation_launch.py headless:=False x_pose:=x y_pose:=y"
```

We also pass in the random generated `x` and `y` coordinates using the `x_pose` and `y_pose` parameters of the node.

After the `nav2_bringup` node is running, we can bring up the `collision_monitor` node:

`run_collision_monitor = "ros2 launch nav2_collision_monitor collision_monitor_node.launch.py"`

This command is running the `collision_monitor` node using the `collision_monitor_node` launch script found in the `nav2_collision_monitor` package. The `nav2_bringup` and `collision_monitor` are run in every experiment as the base simulation; they are followed by the specific node needed for each of the six experiments:

- **LiDAR Passthrough:** This node is run using the following command: `run_lidar_passthrough = "ros2 run lidar_passthrough lidar_passthrough"` In this command, we are running the `lidar_passthrough` ROS node written in Python from the `lidar_passthrough` package. This node subscribes to the `/scan_tom` topic, reading data from the additional LiDAR sensor, and publishes the same messages to the `/scan_fakeobject` topic.

- **LiDAR Passthrough with Odometry Subscription:** Similarly to the LiDAR passthrough experiment, we run `run_lidar_passthrough_odom = "ros2 run lidar_passthrough_odom lidar_passthrough_odom"`

  This node subscribes to both the `/scan_tom` and `/odom` topics and publishes LiDAR messages to the `/scan_fakeobject` topic. An additional experiment is created on top of this called `passthrough_delayed` which runs the

  `lidar_passthrough_odom`

  node thirty seconds after the `nav2_bringup` and `collision_monitor` start.

- **Fake Object:** The fake object experiment employs the fake object node created by Ella to attack the TurtleBot's navigation algorithm by mutating the compromised LiDAR sensor's messages in an intelligent manner where the navigation algorithm believes there is an object at a specific pose on the map [18]. The fake object node subscribes to both `/scan_tom` and `/odom` topics and publishes LiDAR messages to the `/scan_fakeobject` topic, mimicked by the previous LiDAR pass through experiment. To run this node, the following command is executed:

  ```
  run_fake_object = "ros2 run nav2 fakeobject"
  ```

  Additionally, we run the `fakeobject_delayed` experiment with the `fakeobject` node starting thirty seconds after the `nav2_bringup` and `collision_monitor` start

- **Random Noise:** The final experiment mode involves a brute force fuzzing on the LiDAR sensor by introducing random noise to the values in the `/scan_tom` topic. We run the node using the following command:

  ```
  run_random_noise = "ros2 run random_noise random_noise"
  ```

  This command is running the `random_noise` node from the `random_noise` package. The function of the node is simple; it subscribes to the `/scan_tom` topic and publishes to `/scan_fakeobject` after adding random uniform noise to the ranges in the `LaserScan` messages. The ranges array in `LaserScan` represent an array of distances of an object to the sensor at a specific angle [46]. The range values are used predominantly for object avoidance [46].

Each experiment is designed in pairs, aligning malicious tests with corresponding

non-malicious tests to examine structural influences within the ROS system. Specifically, we aim to understand how variations in the system graph structure impact the clustering of experiment runs, observing whether malicious behavior introduces distinguishable patterns. Malicious experiments add additional nodes, creating dependencies within the system. To address this, passthrough nodes are introduced, which replicate the graph structure of the malicious nodes without altering data, isolating structural differences while minimizing the effect of new dependencies.

The passthrough nodes mirror both the structural characteristics and the message delay introduced by the malicious nodes, allowing for more controlled experimentation. The LiDAR passthrough experiment emulates the structural and timing effects of the random noise experiment, while the LiDAR passthrough with an Odometry subscription replicates the setup of the fake object experiment. This setup minimizes the impact of structural differences on the ROS graph, allowing us to focus on the behavioral variations in clustering without the influence of graph structure discrepancies.

`NavigateToPose` Action

When all the nodes - Nav2 specific nodes as well as nodes for each respective experiments - are running, we begin the experiment by sending a ROS Action:

```
run_nav_to_dest = "ros2 action send_goal /navigate_to_pose
nav2_msgs/action/NavigateToPose p̈ose: {header: {frame_id: map}, pose:
{position: {x: -1.5, y: 1.5, z: 0.0}, orientation: {x: 0.0, y: 0.0,
z: 0, w: 1.00000}}}¨
```

A ROS 2 Action is a communication mechanism designed for asynchronous, long-running tasks, allowing clients to send goals, receive periodic feedback, and obtain results upon task completion [39]. It is ideal for operations that require extended execution time, including robot navigation [39].

There are several components in the action seen above. We send a navigation goal to the TurtleBot3 robot using the `NavigateToPose` action from the `nav2_msgs` package [37]. The command sends a goal to the `navigate_to_pose` action server to move the robot to the position (-1.5, 1.5) in the map frame; the target pose provided in the command specifies the `x` and `y` coordinates which can be modified with random values generated in the same method seen in Chapter 5.

Graph Creation - NetworkX

The `run_experiments.py` script contains every command specified in this section. The researcher will pass in two arguments to the script, the experiment mode and the number of runs. The manual for the script is seen below:

```
usage: run_experiments.py [-h] [-m {lidar_passthrough,lidar_passthrough_odom,
fake_object,random_noise, delayed_passthrough,delayed_fakeobject}] [-n NUM_RUNS]


options:
  -h, --help            show this help message and exit

  -m, --mode {lidar_passthrough,lidar_passthrough_odom,fake_object,random_noise,
  delayed_passthrough,delayed_fakeobject}  Experiment mode
```

```
-n NUM_RUNS, --num_runs NUM_RUNS  Number of runs
```

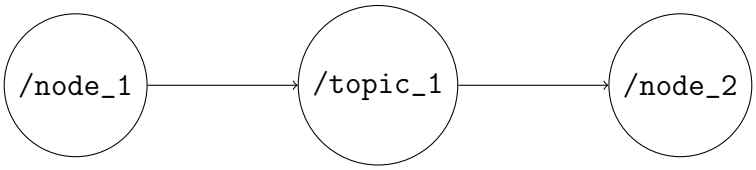
To run the script, we pass in the experiment mode and number of runs as arguments:

```
python3 run_experiments.py -m lidar_passthrough -n 100
```

In the example above, we run the LiDAR Passthrough experiment 100 times. The output of the `run_experiments.py` is a ros bag with all the messages communicated over the course of the experiment, as well as a graph representing the communication between topics within the ROS network.

In order to create and save the ROS network graph, we use the NetworkX Python package [27]. NetworkX is a Python package used for the creation, manipulation, and study of complex networks or graphs [27]. It provides tools to work with both simple and complex graph structures, including nodes, edges, and attributes, and includes algorithms for analyzing network properties, such as shortest paths, clustering, and connectivity [27].

While we are running the experiments described above, we create a graph using the NetworkX package to create graph nodes and edges between these nodes. Each node in the graph represent a ROS topic or ROS node and each edge represents communication from a ROS node to a ROS topic. For example, if we have two nodes and one topic - `/node_1`, `/node_2`, and `/topic_1` - where `/node_1` publishes to `/topic_1` and `/node_2` subscribes to `/topic_1`, the graph will look like the following:



With large numbers of ROS nodes and topics, this graph becomes very complex hence the need to use the NetworkX package. The graph represent the full experiment is created in two steps:

1. Run `ros2 node list` to retrieve a list of all ROS2 nodes currently running; add each ROS node to the graph as a node

2. On each node from step 1, run `ros2 node info <node>` [39]. This command returns a list of topics which the given node subscribes and publishes to; add each topic to the graph as a node and add each edge between the ROS node and topic to the graph

At the end of the experiment, the graph is saved in GEXF format to be used in the succeeding steps. GEXF (Graph Exchange XML Format) saves the graph information in an XML format which can be imported by the NetworkX package in future scripts to extract features [27].

Resources Used

A total of six experiments were conducted using a Lenovo ThinkPad equipped with the following hardware specifications: Intel Core i7-2630QM CPU, 32GB of RAM, 500GB of internal storage, and a 2TB external hard drive. During each run, the graph produced was divided into 10-second intervals. With an average run duration of 3 to 4 minutes, this interval size resulted in approximately 18 subgraphs per run, which was a manageable number for analysis. Each ROS bag recorded per experiment occupied an average of 1GB of storage. Given the substantial data size for each run of the experiment and the limited resources of the system, the total number of runs was limited to 15 per experiment.

5.4 Extracting Feature Vectors

After the bags and graphs have been created, we need to extract information from these recorded messages and graph structure in order to cluster the data and find patterns. As discussed previously, agglomerative heirarchical clustering is the most effective clustering algorithm in the given experiment [8]. Agglomerative heirarchical clustering requires a distance calculation in order to cluster similar data points, thus we extract features determined to be relevant to finding malicious experiments. The features are then added to an N-dimensional vector to be used by the clustering algorithm to find patterns between experiments.

We can separate the extracted features into two categories; structural features and contextual features.

Structural Features

Structural features are extracted from the ROS network and utilise graph-based algorithms to represent the structure of the ROS communication during the experiment. Motivation for extracting graph based features is seen in the Malchain framework and RoboFuzz framework, leading to very successful results [23, 32]. In the Malchain framework, the authors applied a clustering algorithm on feature vectors derived from the system call graph between microservices. The features of which the vectors were comprised of are entropy, variance, and flow proportion as the local features and closeness centrality and betweenness centrality as the global feature [23]. In our framework, the following structural features were extracted:

- **Max Degree**

  Degree of a node in graph theory refers to the number of edges connected to the

given node [44]. Using the NetworkX package, the degree of every node in the graph can be easily accessed [27]. The maximum degree is the highest number of edges any one node connects to.

- **Average Degree**

  The average degree of all nodes in the ROS communication graph. Using both the maximum degree and average degree, we can abstract information regarding the network connectivity in the ROS graph.

- **Density**

  Density is the ratio between edges in a graph and the maximum possible edges [44], which can also be calculated using the NetworkX package [27]. When used for clustering, density helps to identify groups of nodes that are more interconnected.

- **Entropy**

  Entropy measures the uncertainty or disorder in the distribution of node degrees within the graph [44]. We first calculate the probability distribution of node degrees in the graph. Using these probabilities, we calculate entropy using the following formula:

$$H = -\sum_{i=1}^{n} p_i \log p_i$$

  [11] where p_i represents the degree probability for the i-th degree. The entropy measure provides a way to quantify how evenly the connections are spread across the network [11]

- **Variance**

  Additionally, we calculate the variance of the node degrees in order to help

distinguish different graph structures. In the malicious graph which contain the extra fuzzing node, the degree variance will be slightly different compared to a normal experiment run without any additional nodes.

- **Max Centrality**

  Using NetworkX, we can also calculate the in-degree centrality for each node, which is the fraction of nodes its incoming edges are connected to [27]. A node with a high in-degree centrality indicates the nodes receives many messages and thus is a key ROS node or topic during the experiment. The maximum centrality extracts the node with the highest in-between centrality.

- **Average Centrality**

  From the previously calculated in-degree centrality, we also calculate the average centrality across all nodes in the graph.

Contextual Features

We use the graph to extract information regarding the structure of the ROS network graph. Additionally, we can leverage the context of the given problem domain to extract behavioural features. Extracting contextual features can greatly aide in clustering based off behaviour however, it requires domain knowledge in order to craft useful feature vectors. In our experiments, we extract the following contextual feature vectors:

- **Variance of Yaw**

  Yaw is one of the three Euler angles that describe the TurtleBot3's orientation in 3D space [52]. Yaw specifically refers to the robot's rotation around it's vertical axis, which is perpendicular to the ground [52]. If the variance of the yaw

is high, then we can assume the robot is frequently turning, performing erratic or unstable movements, or navigating a challenging environment. All of these assumptions are expected in experiments with compromised sensors thus we expect the variance to be higher in malicious experiments compared to regular experiments.

We can retrieve the yaw value by reading messages from the `/odom` topic and storing every `orientation` value throughout the course of the experiment. The data in `orientation` are quaternions so we need to convert them to Euler [39]. The conversion is done using the following algorithm [54]:

```
def quat2eulers(q0:float, q1:float, q2:float, q3:float) -> tuple:
    roll = math.atan2(
        2 * ((q2 * q3) + (q0 * q1)),
        q0**2 - q1**2 - q2**2 + q3**2
    )
    pitch = math.asin(2 * ((q1 * q3) - (q0 * q2)))
    yaw = math.atan2(
        2 * ((q1 * q2) + (q0 * q3)),
        q0**2 + q1**2 - q2**2 - q3**2
    )
    return (roll, pitch, yaw)
```

Once we have collected all yaw values throughout the course of the experiment, we calculate the variance and append the result to the feature vector.

- **Average Range Variance**

  Each message published to the topic `/scan` is of type `/LaserScan` [37]. Each `/LaserScan` message includes a `ranges[]` attribute which captures the distance from an object at each given angle [37]. Clustering using the feature representing the average range variance for an experiment can help group experiments in similar environments together. When a fake object is introduced, the `ranges[]` value would indicate an obstacle is near and modify the variance in comparison to a regular experiment run.

- **Duration of Navigation**

  In experiments consisting of many obstacles on the TurtleBot's navigation path, the total duration increases as the navigation algorithm recalculates the path to the destination [37]. When a fake object is added, the experiment can potentially be consistently blocked while the navigation algorithm attempts to find a valid path to the target pose

Plugin Architecture

As seen in the RoboFuzz framework, building an oracle to accurately represent the expected behaviour requires significant domain knowledge [32]. In our framework, we overcome this issue by developing the feature vector extraction process using a plugin architecture. Using this approach, users are able to trivially add and remove feature extraction plugins. The structure of the program is the following:

```
extract_vectors/
|
+-- main.py
```

```
+-- plugin_manager.py
+-- plugin_interface.py
+-- config.yaml
+-- plugins/
    |
    +-- structural_features.py
    +-- contextual_features.py
```

All plugins can be found in the `plugins/` directory and must implement the `plugin_interface`. The `plugin_interface` requires the new plugin to implement the `extract_features(self, G, cur)` method, where input parameter `G` is the NetworkX graph and `cur` is a cursor to the database containing recorded ROS messages from the ROS bag. Using either or both graph and experiment messages, users have the ability to create a wide range of feature extraction plugins. The `plugin_manager` then reads all plugins from the plugins directory specified in `config.yaml` and extracts features based on the `extract_features` method in each plugin.
The proposed plugin architecture demonstrates the simplicity in adding and removing plugins with a wide variety of data, network graph and ROS messages, needed to extract relevant features that accurately model the experiments.

5.5 Clustering

After extracting feature vectors, we apply a clustering algorithm to determine whether there are noticeable patterns across experiments such that malicious and normal experiments can be distinguished. We want to apply the agglomerative heirarchical clustering algorithm as described in [8].

To perform agglomerative heirarchical clustering we employ the Scipy library, a Python library built on top of NumPy that contains a collection of mathematical algorithms [51]. Before we can apply the clustering algorithm, we clean, calculate the distance matrix, and bootstrap the feature vector dataset. When exporting the extracted feature vectors to a dataset, the feature vector is stored as a string. We first replace all null values to "0" and convert the features to integers. Once we have the numerical vector of features, we can calculate the distance matrix. Due to limited storage space, the number of experiments which we can perform is low. We can bootstrap the collected experiment data to increase the number of data points used for clustering. Bootstrapping is the process of repetitively resampling the data with replacement in order to gather a larger dataset.

In agglomerative hierarchical clustering, the distance matrix is essential because the algorithm clusters data based on the proximity of points; the distance matrix is used to construct a dendrogram, a tree-like structure that visually represents the merging of clusters at different levels [51]. To calculate the distance matrix, the SciPy library provides the `pdist` function. `pdist` calculates the pairwise Euclidean distance between points in an N-dimensional space using the given algorithm: [51]

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

where x and y are two points in the N-dimension vector space. Note that feature vectors must all be of equal length.

We then apply the `linkage` function provided by SciPy to the distance matrix, performing heirarchical agglomerative clustering [51]. The `linkage` function iteratively merges pairs of clusters based on the selected distance criterion [51]. The `linkage` function outputs a heirarchical tree (dendogram); this output is passed to the `fcluster` SciPy function in order to extract flat clusters from the dendogram [51]. `fcluster` allows us the define a threshold to extract flat clusters from a given level of the dendogram [51]. To determine the best parameters for heirarchical agglomerative clustering, we perform a grid search. Grid search is the process of testing all permuation of parameters and choosing the best combination based on a metric, in this case accuracy. Since we know which experiments are malicious and non-malicious, we can calculate the accuracy of the clustering by determining which cluster has a larger percentage of correctly identified malicious feature vectors. We also note the Area Under the Curve (AUC) of each clustering algorithm. AUC is commonly used to measure the efficacy of clustering algorithms [31]. AUC is a metric used to evaluate the performance of models by measuring the area under the ROC (Receiver Operating Characteristic) curve. It indicates how well a model can differentiate between positive and negative classes, in our case malicious and non-malicious experiments [31]. AUC ranges from 0 to 1, where 1 signifies perfect classification, 0.5 represents random guessing, and values below 0.5 suggest the model is performing worse than random [31].

The `linkage` SciPy function offers several linkage methods; our grid search applies the following four algorithms [51]:

1. **Ward**

   This linkage method applies the Ward variance minimization algorithm, that

merges clusters in a way that minimizes the total variance within all clusters [51]. At each step, it finds the pair of clusters that when combined, results in the smallest increase in the overall variance [51]. The following formula is applied:

$$d(u,v) = \sqrt{\frac{|v|+|s|}{T}d(v,s)^2 + \frac{|v|+|t|}{T}d(v,t)^2 - \frac{|v|}{T}d(s,t)^2}$$

where u is the new cluster combining clusters s and t and $T = |v| + |s| + |t|$ [51]

2. **Single**

   The single method applies the Nearest Point Algorithm, as follows:

$$d(u,v) = min(dist(u[i], v[j]))$$

   where i are all points in cluster u and j are all points in cluster v [51]

3. **Complete**

   Similar to the single method, the complete method applies the Farthest Point Algorithm [51]:

$$d(u,v) = max(dist(u[i], v[j]))$$

4. **Average**

   The average method applies the UPGMA (Unweighted Pair Group Method with Arithmetic Mean) algorithm, which iteratively merges the closest clusters based on their distance and recalculating the distances using the arithmetic mean [51].

The applied algorithm looks like the following [51]:

$$d(u, v) = \sum_{ij} \frac{d(u[i], v[j])}{|u| * |v|}$$

The second parameter we perform grid search on is the threshold criterion to create flat clusters for the `fcluster` function [51]. The tested values are the following:

1. `distance`

   Creates flat clusters such that the similarity distance (calculated by the `linkage` function) between any two experiments within the same cluster does not exceed the given threshold [51]

2. `maxclust`

   Identifies the smallest threshold r such that the similarity distance between any two experiments in the same flat cluster is at most r, while ensuring that no more than the given number of flat clusters are created (two in our case) [51]

The results of the grid search are presented in the table below:

Linkage Algorithm	**Threshhold Criterion**	**Accuracy**	**AUC**
Ward	`distance`	0.029	0.514
Ward	`maxclust`	0.953	0.498
Single	`distance`	0.027	0.513
Single	`maxclust`	0.924	0.484
Complete	`distance`	0.027	0.513
Complete	`maxclust`	0.924	0.484
Average	`distance`	0.924	0.484
Average	`maxclust`	0.027	0.513

Table 5.1: Grid Search Results for Clustering Parameters

Based on the results from the grid search, the optimal parameters chosen are Ward's distance for the linkage algorithm, and `maxclust` threshhold criterion for `fcluster`.

Chapter 6

Results

The following section outlines the results of experiments designed to assess the efficacy of a clustering based oracle on navigation experiments simulated using ROS2.

As discovered in the previous section, the optimal parameters used to perform clustering are utilising Ward's distance for the `linkage` algorithm and the `maxclust` threshold criterion for `fcluster`.

6.1 Graph Results

he ROS system graphs generated from each experiment were comprised of hundreds of interconnected nodes and edges. On average, these graphs contained over 100 nodes and more than 300 edges. In these representations, the nodes correspond to both ROS nodes and topics, while the edges depict the message exchanges between them. An example of such a graph is shown below:

This graph is challenging to interpret visually due to its complexity. However, the primary focus of our experiments lies in analyzing the topology of specific nodes and topics.We use feature vectors to abstract and represent the relevant information from the graph.

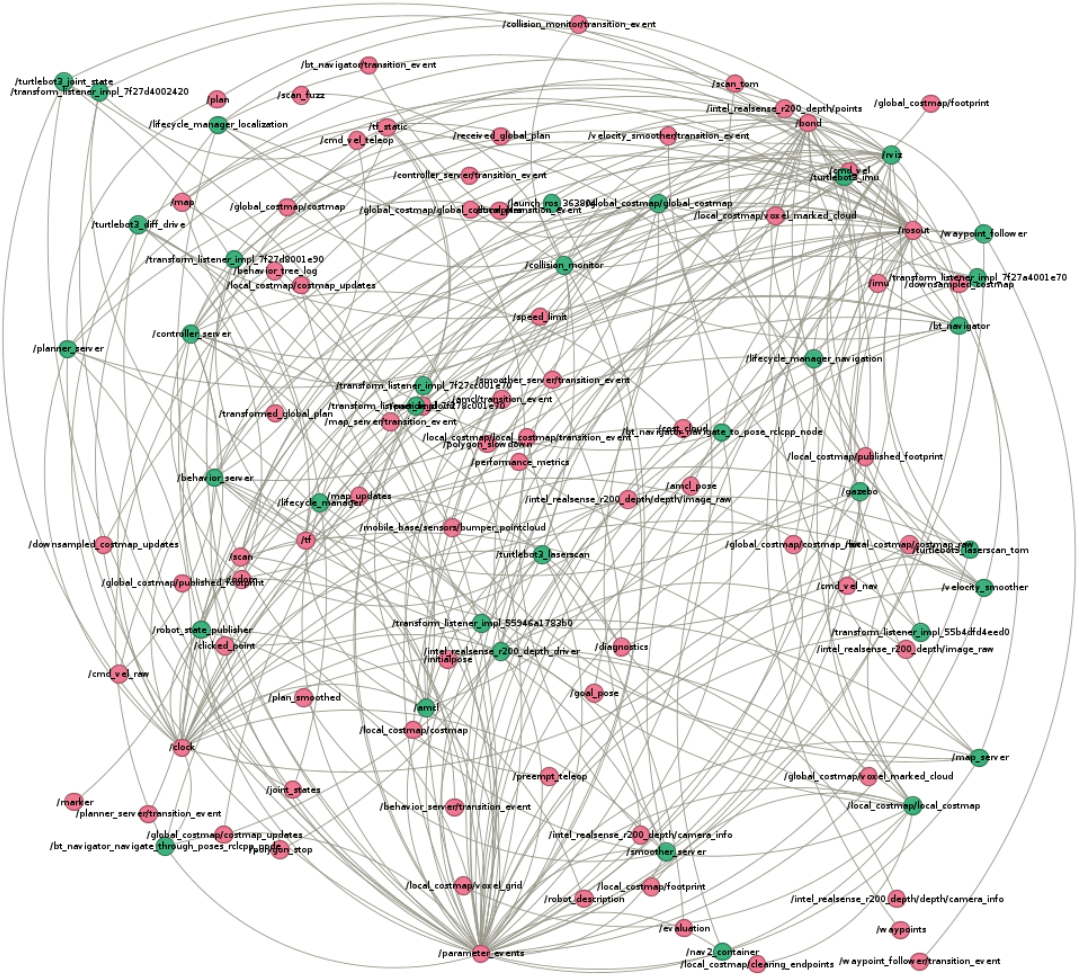


Figure 6.1: ROS system graph example. Green nodes are ROS nodes and red nodes are ROS topics.

In the paired experiments of 'LiDAR Passthrough' against 'Random Noise', an additional node, `/scan_fakeobject`, is introduced. In the 'LiDAR Passthrough' experiment, this node functions as a passthrough, forwarding unaltered messages to the

`/collision_monitor` topic. This topic sends the messages to nodes involved in collision monitoring, such as `/polygon_slowdown`, which generates a polygon around the TurtleBot3 simulation. This polygon slows down the robot if obstacles are detected within its boundaries [37].

In the 'Random Noise' experiment, the same node with the same topology is created; however, its behavior differs. The node mutates the `ranges` values in the messages by introducing random noise, relaying incorrect information to the subscribers of the `/collision_monitor` topic. The intent of this setup is to ensure that both experiments share an identical structural configuration, isolating the impact of message alterations on system behavior.

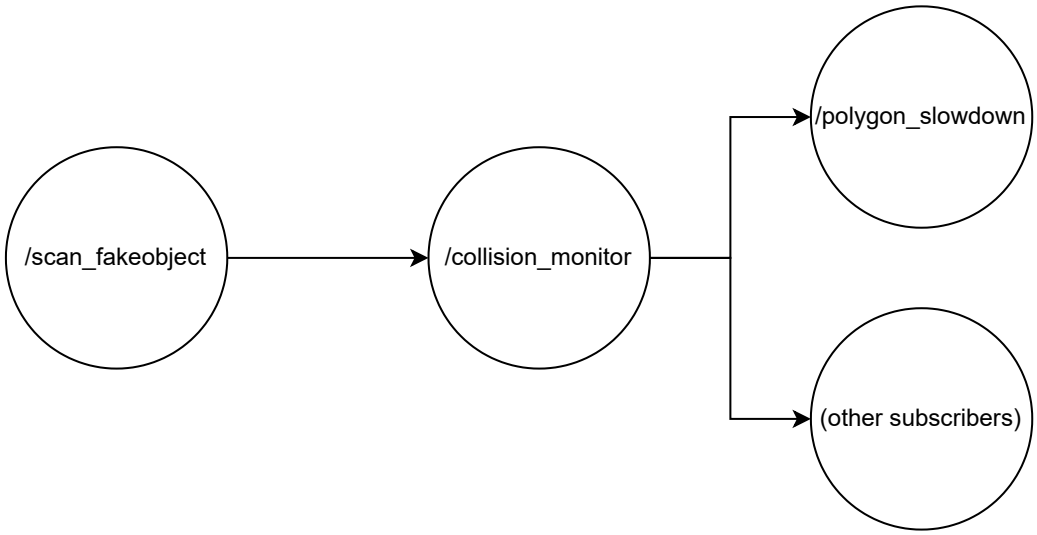


Figure 6.2: Graph of LiDAR Passthrough and Random Noise Experiments

Similarly, in the paired experiments of 'LiDAR Passthrough with Odometry Subscription' against 'Fake Object', the `/scan_fakeobject` malicious node is introduced with an additional subscription to the `/odom` topic. In the malicious 'Fake Object'

experiment, this node mutates data received from the `/odom` topic and publishes it to the collision monitor, global costmap, and local costmap topics. These topics are subscribed to by nodes responsible for providing the robot with essential perception information for navigation, as detailed in Chapter 5 [4].

In contrast, the 'LiDAR Passthrough with Odometry Subscription' experiment replicates the structure of the malicious experiment without altering the data.

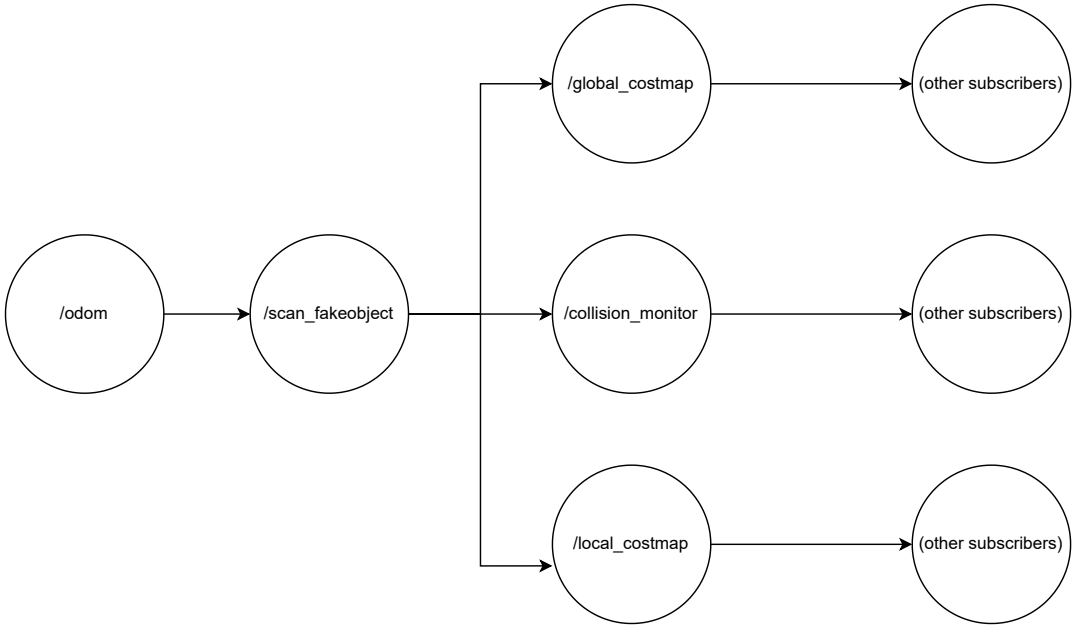


Figure 6.3: Graph of LiDAR Passthrough with Odom and Fake Object Experiments

6.2 Overall Clustering Results

We bootstrap the small dataset of experiment runs to simulate 1000 experiments where 30% are malicious. We apply the agglomerative heirarchical clustering algorithm on these feature vectors to divide the experiments into two clusters. The

optimal algorithm resulted in a 95.3% accuracy but an AUC value of 0.504. Adjusting the rate of malicious experiments slightly improves the AUC and accuracy as seen in the table below:

Malicious Experiment Percentage	**Accuracy**	**AUC**
10%	94.1%	0.496
30%	95.3%	0.504
50%	92.4%	0.490
70%	97.0%	0.502
90%	97.1%	0.506

Table 6.1: Grid Search Results for Rate of Malicious Experiments

As seen in Table 6.1, the accuracy is consistently high while the AUC remains around 0.5. This indicates that the clustering algorithm does a good job at predicting the majority class but is fails to distinguish malicious and non-malicious experiments.

The dendogram seen in 6.4 represents the hierarchical clustering of the complete experiment data. The y-axis represents the distance between clusters during the clustering process. As the scale is very large, variations in cluster similarity is indicated. The initial split, seen in the large blue cluster, shows a large split in experiments into two groups, with finer separations happening later down the dendogram in the orange and green clusters. The initial blue grouping can indicate major dissimilarity between malicious and non-malicious experiments which is supported by the strong accuracy. However, the low AUC value indicates that malicious and non-malicious experiments may be tightly coupled in the lower orange and green clusters.

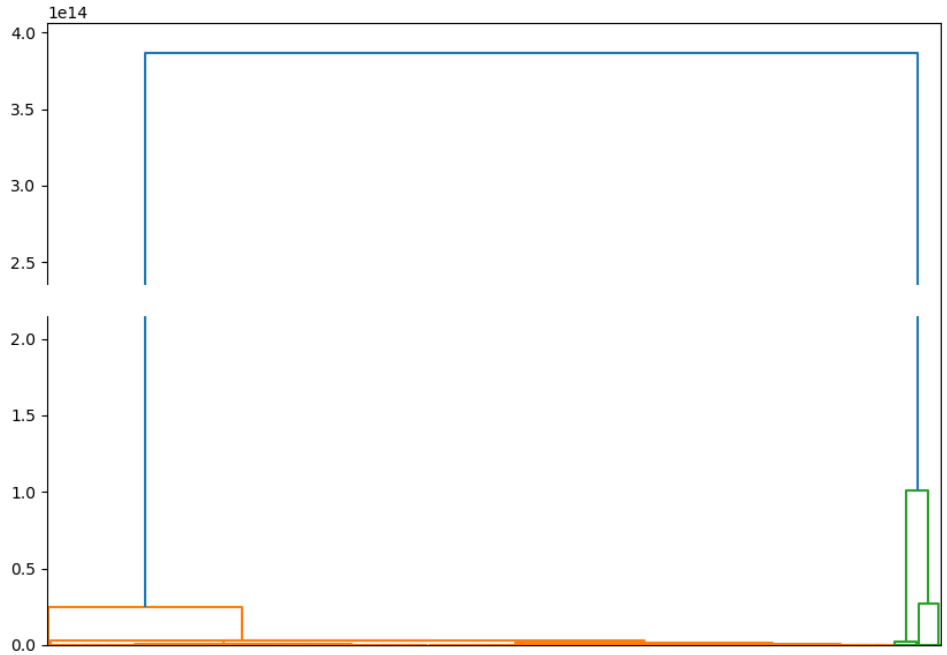


Figure 6.4: Dendogram of Overall Clustering Experiment

6.3 Clustering on Contextual Features

When clustering on solely on the contextual features, the results are similar to the overall clustering results, also seen in the dendogram in Figure 6.5. We achieve an accuracy of 95.3% with an AUC of 0.499.

6.4 Clustering on Structural Features

Clustering on the structural features in the feature vectors produced an accuracy of 78.1% with an AUC of 0.711. While the accuracy is lower, the large improvement in AUC suggests that clustering on solely structural features allows the model to better

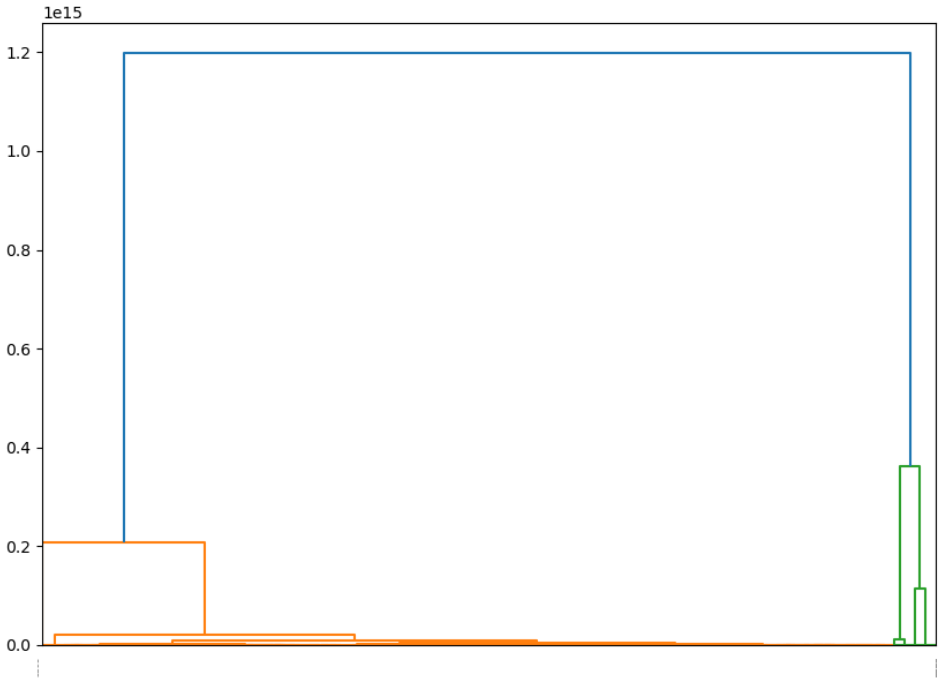


Figure 6.5: Dendogram of Contextual Clustering Experiment

classify positive and negative experiments.

The dendogram for this experiment in Figure 6.6 shows signifcant differences from the previous two experiments. The height of the blue cluster is much less indicating a greater degree of similarity between experiments. Overall, clustering structural features lead to tighter groups and more similarity between clusters. This can be explained by the reduction of noise from contextual features, leading to a better AUC score as well.

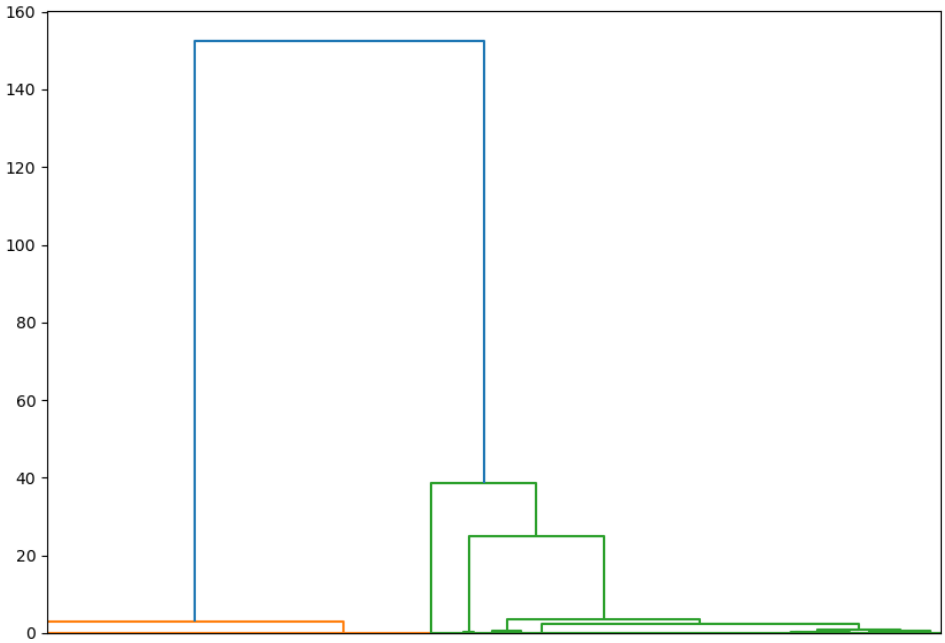


Figure 6.6: Dendogram of Structural Clustering Experiment

6.5 Pairwise Clustering Results

Clustering paired experiments offers valuable insights into the effectiveness of the clustering algorithm in distinguishing between malicious and normal experiment runs.

LiDAR Passthrough against Random Noise

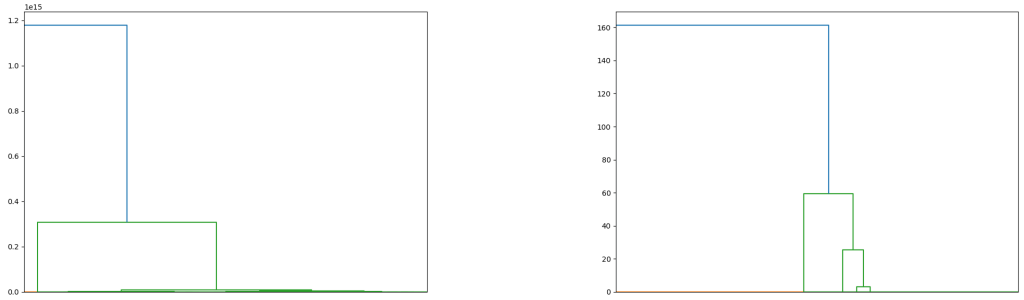
The experiment involving the LiDAR passthrough with random noise achieved an accuracy of 93.8% and an AUC (Area Under the Curve) of 0.533, consistent with the results observed in overall clustering. These findings suggest that while the clustering

Experiment	Accuracy	AUC
LiDAR Passthrough/Random Noise (all features)	93.8%	0.533
LiDAR Passthrough/Random Noise (structural features)	100%	0.9669
LiDAR Passthrough with Odom/Fake Object (all features)	93.4%	0.4672
LiDAR Passthrough with Odom/Fake Object (structural features)	100%	1.0000
Delayed LiDAR Passthrough with Odom/-Fake Object (all features)	86.6%	0.4662
Delayed LiDAR Passthrough with Odom/-Fake Object (structural features)	66.4%	0.8318

Table 6.2: Pairwise Clustering Results

algorithm performs well in identifying patterns within the dataset, its ability to distinguish malicious behavior from normal runs using all features remains ineffective. When clustering was performed on solely structural features, the accuracy improved to 100%, and the AUC increased to 0.9669. These results highlight the effectiveness of focusing solely on structural features, as they appear to provide clearer distinctions between malicious and normal experiment runs. The sharp improvement underscores the importance of the ROS messaging graph structure in driving the clustering results.

Achieving 100% accuracy is not entirely conclusive on its own. There are multiple factors that could contribute to this outcome, such as overfitting to specific structural patterns or an imbalanced dataset. However, the high AUC value of 0.9669 is a



((a)) Contextual Features ((b)) Structural Features

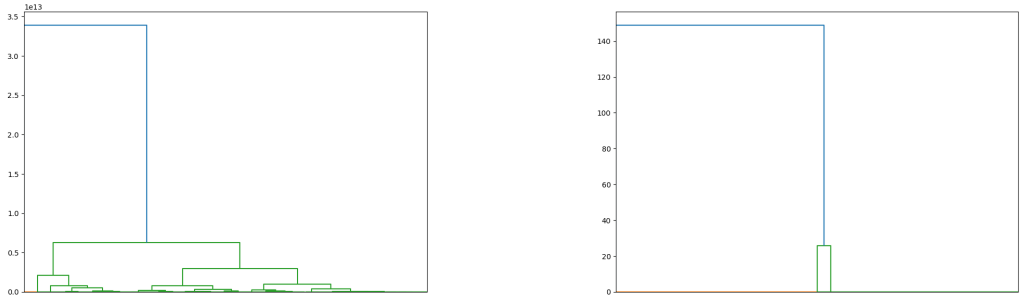
Figure 6.7: Dendograms from Lidar Passthrough against Random Noise Experiment

strong indicator of the effectiveness of the clustering algorithm when using structural features. An AUC close to 1 reflects a strong ability of the model to distinguish between positive and negative classes at varying thresholds [31]. This further suggests that structural aspects of the ROS graph play a critical role in identifying and distinguishing different experimental conditions.

LiDAR Passthrough with Odometery Subscription against Fake Object

The overall clustering results for this experiment yielded an accuracy of 93.4% and an AUC of 0.4672. This aligns with the observed pattern of high accuracy but a relatively poor AUC, indicating a potential imbalance in the clustering algorithm's performance. Although high accuracy suggests that the algorithm successfully classifies most of the experiments, the low AUC highlights its struggle to differentiate effectively between true positives, false positives, true negatives, and false negatives.

When clustering was performed exclusively on structural features, the results were seemingly perfect, with 100% accuracy and an AUC of 1.0. This result raises questions about the generalizability of the model. It suggests that the structural features may



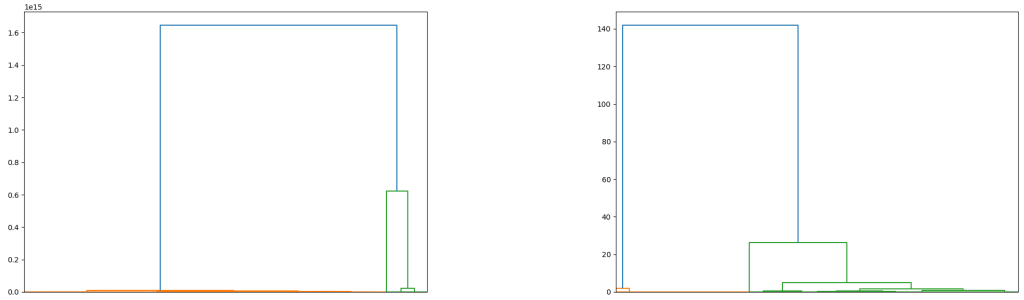
((a)) Contextual Features ((b)) Structural Features

Figure 6.8: Dendograms from Lidar Passthrough with Odom against Fake Object Experiment

completely define the clustering, potentially leading to overfitting. This outcome could mean that the clustering algorithm is overly reliant on structural differences that exist in the dataset but might not capture the actual behavioral nuances or relationships between malicious and normal experiment runs.

The delayed experiment involving the LiDAR passthrough with Odometry subscription and fake object produced interesting results, with a lower accuracy of 86.6% and an AUC of 0.4662. These results indicate a decline in the model's overall performance compared to other experiments, suggesting that the delayed start introduced complexities that made it more difficult for the clustering algorithm to maintain high accuracy while distinguishing between malicious and regular experiment runs.

When the clustering was done on structural features, the accuracy dropped to 66.4%, but the AUC improved to 0.8318. This outcome highlights the challenges introduced by the delayed start of malicious behavior, which obscured the structural distinctions between malicious and regular experiments. This result contrasts the previous two experiments where the accuracy increased when clustering on structural features, suggesting that the delayed initiation of the malicious object disrupted the



((a)) Contextual Features ((b)) Structural Features

Figure 6.9: Dendograms from Delayed Lidar Passthrough with Odom against Fake Object Experiment

clustering model's ability to group malicious experiments with their corresponding patterns effectively. The temporal lag appears to have added a layer of complexity, reducing the effectiveness of structural features and making it more challenging for the algorithm to maintain high performance.

6.6 Suggestions for Improvement

Previously, we see that structural features provide a better model compared to contextual features for the experiments. Additional methods that we can employ to improve the low AUC results are the following:

- **Adjust Sampling**

  One common cause for a low AUC is a large class imbalance in the dataset. Creating and recording more malicious experiments can help address the low percentage and variance of the minority class in our dataset.

- **Adjust Evaluation Metric**

  When applying grid search to find optimal parameters, we chose parameters

based on the accuracy. While this satisfies the overall goal of our research, the resulting model is not able to effectively distinguish between positive and negative class. Performing grid search with metrics such as AUC, precision, recall or F1 score might provide a better model.

- **Use More Complex Model** While agglomerative heirarchical clustering is the suggested algorithm seen in [8], more complex models might provide a better oracle. Potential clustering algorithms include decision trees, random forests or gradient boosting.

6.7 Threats to Validity

Internal Threats

The internal validity of the study is influenced by several factors. First, the characteristics of the experimental setup restricted the range of results obtained, potentially limiting the depth and breadth of the insights.

Additionally, dataset size constraints, imposed by resource and storage limitations, may have affected the robustness of the findings, leaving certain aspects underexplored. It is important to note that the size of the data set poses storage challenges. Each experiment generates multiple communication graphs segmented by time intervals, along with a ROS bag containing all messages and metadata recorded during the run. On average, each experiment consumes approximately 5GB of storage space, leading to the rapid utilization of a 2TB external hard drive. To further enhance the experimental results, additional data collection is necessary, which inherently will demand even greater storage and resource capacity.

Furthermore, while the experimental methodology allowed for flexibility, incorporating a more structured setup could be a consideration for future work. In this research, the starting and ending points for each experiment were randomly selected to reflect realistic conditions. While this approach introduced variability, it aligns with the goal of studying performance under diverse scenarios. Future investigations could explore the impact of using more controlled setups to complement the findings of this study without requiring significant changes to the current experimental design.

External Threats

Several external factors threaten the generalizability of the study's outcomes. The autonomous navigation domain is rapidly evolving, requiring increasingly sophisticated domain expertise to develop effective evaluation oracles. This may reduce the effectiveness of a clustering based oracle in grouping complex experiments.

Furthermore, the limited size of the data set increases the risk of overfitting, reducing the model's ability to generalize to novel scenarios. Reliance on the ROS framework also presents a threat to generalization, as findings may not be directly transferred to systems developed using alternative architectures or frameworks, or if the industry standard moves away from ROS.

Chapter 7

Summary and Conclusions

7.1 Summary

Fuzz testing frameworks tend to be limited as they focus primarily on identifying software crashes and memory safety issues, often missing more subtle or complex bugs. Additionally, the development of a suitable oracle, which is necessary to distinguish between successful and failed tests, demands significant domain expertise. This makes it challenging to create a generalized approach to fuzz testing in ROS environments.

In this thesis, we present a testing framework aimed at addressing the oracle problem through clustering techniques. The dataset comprises multiple experimental runs, both malicious and non-malicious, within a ROS2 environment using a TurtleBot3 autonomous navigation simulation. From these runs, feature vectors that capture the structural and contextual characteristics of the ROS2 system are extracted and clustered using agglomerative hierarchical clustering. The framework is organized into the following steps:

1. Record Experiments

2. Build Graph to Model Communication

3. Extract Feature Vectors from Graphs and Messages

4. Cluster Vectors

The extraction program is designed with a plugin architecture. This modular approach allows for the development of both general and specific features, utilizing either ROS message recordings via `rosbags` or the ROS network graph generated by the NetworkX Python library. This flexibility enables easy integration of new features without modifying the core system, making the framework adaptable to various use cases and system configurations.

The following experiments were simulated using the TurtleBot3 autonomous navigation using Nav2 in ROS2:

- Experiment 1: Normal run with LiDAR passthrough
- Experiment 2: Normal run with LiDAR passthrough and subscription to topic `/odom`
- Experiment 3: Malicious run with random noise added to LiDAR sensor
- Experiment 4: Malicious run with fake object perceived by sensors [18]
- Experiment 5: Normal run with delayed start of LiDAR passthrough and subscription to topic `/odom`

  Experiment 6: Malicious run with delayed start of fake object node

From these experiments, structural and contextual feature vectors were extracted. The extracted structural features from the ROS network graph are:

- Max degree of any node

- Average degree of all nodes
- Density
- Entropy of node degrees
- Variance of node degrees
- Max centrality of any node
- Average centrality of all nodes

The extracted contextual features from the ROS messages are

- Variance of yaw
- Average range variance
- Duration of navigation

The extracted features are then clustered using an agglomerative heirarchical clustering model. The clustering results demonstrated high accuracy in detecting malicious experiments, with accuracy rates often exceeding 92%. However, other metrics, such as AUC, indicated a high incidence of false positives. When using only structural features for clustering, the accuracy was lower, but the model exhibited improved ability to distinguish between malicious and non-malicious experiments.

7.2 Future Work

There are several promising directions for future work. Due to resource constraints, the current dataset of experiments is quite limited. Expanding the dataset by including a wider variety of experiment types and increasing the number of runs would

enhance the framework's effectiveness and robustness.

As previously discussed, while the clustering results achieved high accuracy, they also exhibited a high rate of false positives. To address this, future work could focus on reducing class imbalance in the experiments, adjusting the sampling rate, and refining the evaluation metrics to specifically target false positive reduction. Additionally, exploring more advanced clustering techniques, such as decision trees, random forests, or gradient boosting, may further improve model performance.

The plugin-based architecture of the feature extraction program offers many opportunities for enhancement. By incorporating more sophisticated features that better capture the characteristics of simulations and environments, the framework could become even more accurate. Furthermore, the current implementation is written solely in Python, whereas many ROS systems are developed in C++. Adding C++ support would broaden the framework's accessibility, enabling more developers to utilize and contribute to its development.

7.3 Conclusion

Robotic systems are increasingly being integrated into domains such as healthcare, autonomous vehicles, and industrial automation, where safety, reliability, and performance are critical. As these systems become more complex and autonomous, testing methods must evolve to address domain-specific issues that can emerge from real-world interactions. An effective testing framework is essential to ensure that robotic systems can operate safely and correctly in dynamic and unpredictable environments. By improving the robustness of testing in robotic systems, we can better ensure that

these technologies meet the high standards required for their deployment in safety-critical applications.

Bibliography

[1] xacro. https://github.com/ros/xacro/wiki.

[2] Afl (american fuzzy lop). https://afl-1.readthedocs.io/en/latest/, 2019.

[3] Sdformat. http://sdformat.org/, 2020.

[4] Costmap 2d. https://docs.nav2.org/configuration/packages/configuring-costmaps.html, 2023.

[5] urdf. https://wiki.ros.org/urdf, 2023.

[6] Automatic dependent surveillance - broadcast (ads-b). https://www.faa.gov/about/office_org/headquarters_offices/avs/offices/afx/afs/afs400/afs410/ads-b, 2024.

[7] Mohannad Alhanahnah. Software quality assessment for robot operating system, 2020.

[8] Rafig Almaghairbe and Marc Roper. Building test oracles by clustering failures. pages 3–7, 05 2015.

[9] Jia-Ju Bai, Hao-Xuan Song, and Shi-Min Hu. Multi-dimensional and message-guided fuzzing for robotic programs in robot operating system. In *Proceedings of*

the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS '24, page 763–778, New York, NY, USA, 2024. Association for Computing Machinery.

[10] Marcel Böhme, László Szekeres, and Jonathan Metzman. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1621–1633, New York, NY, USA, 2022. Association for Computing Machinery.

[11] PA Bromiley, NA Thacker, and E Bouhova-Thacker. Shannon entropy, renyi entropy, and information. *Statistics and Inf. Series (2004-004)*, 9(2004):2–8, 2004.

[12] Elliot Burghardt, Daniel Sewell, and Joseph Cavanaugh. Agglomerative and divisive hierarchical bayesian clustering. *Computational Statistics & Data Analysis*, 176:107566, 2022.

[13] Shuo Chen, Jun Xu, and Emre C. Sezer. Non-Control-Data attacks are realistic threats. In *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD, July 2005. USENIX Association.

[14] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (redos) in practice: an empirical study at the ecosystem scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 246–256, New York, NY, USA, 2018. Association for Computing Machinery.

[15] Bernhard Dieber, Benjamin Breiling, Sebastian Taurer, Severin Kacianka, Stefan Rass, and Peter Schartner. Security for the robot operating system. *Robotics and Autonomous Systems*, 98:192–203, 2017.

[16] Vincenzo DiLuoffo, William R Michalson, and Berk Sunar. Robot operating system 2: The need for a holistic security approach to robotic architectures. *International Journal of Advanced Robotic Systems*, 15(3):1729881418770011, 2018.

[17] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016, pages 110–121. Institute of Electrical and Electronics Engineers Inc., August 2016. Publisher Copyright: © 2016 IEEE.; 2016 IEEE Symposium on Security and Privacy, SP 2016 ; Conference date: 23-05-2016 Through 25-05-2016.

[18] Ella Duffy. A dsl for modifying data in ros systems, April 2024. Available at `https://qspace.library.queensu.ca/items/75ba5a52-50af-4616-94f2-1f42be24cdf5`.

[19] Eve Edelson. The 419 scam: information warfare on the spam front and a proposal for local filtering. *Computers & Security*, 22(5):392–401, 2003.

[20] Esteve Fernandez, Tully Foote, William Woodall, and Dirk Thomas. Next-generation ros: Building on dds. Technical report, ROSCon Chicago, September 2014.

[21] Peach Protocol Fuzzer. Peach protocol fuzzer. `https://github.com/MozillaSecurity/peach`, 2020.

[22] Dominic Gates. Q&a: What led to boeing's 737 max crisis. *The Seattle Times*.

[23] Mohammad Mahdi Ghorbani, Fereydoun Farrahi Moghaddam, Mengyuan Zhang, Makan Pourzandi, Kim Khoa Nguyen, and Mohamed Cheriet. Malchain: Virtual application behaviour profiling by aggregated microservice data exchange graph. In *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 41–48, 2020.

[24] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft. *Queue*, 10(1):20–27, jan 2012.

[25] Rahul Gopinath, Philipp Görz, and Alex Groce. Mutation analysis: Answering the fuzzing challenge, 2022.

[26] Rahul Gopinath and Eric Walkingshaw. How good are your types? using mutation analysis to evaluate the effectiveness of type annotations. pages 122–127, 03 2017.

[27] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[28] Shan He, Tao Song, and Xinkai Wu. *An Improved Adaptive Monte Carlo Localization (AMCL) for Automated Mobile Robot (AMR)*, pages 171–181.

[29] Aki Helin. Radamsa. `https://gitlab.com/akihe/radamsa`, 2019.

[30] Saúl Hidalgo. nanoxml. `https://github.com/saulhidalgoaular/nanoxml`, 2020.

[31] Jin Huang and C.X. Ling. Using auc and accuracy in evaluating learning algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 17(3):299–310, 2005.

[32] Seulbae Kim and Taesoo Kim. Robofuzz: fuzzing robotic systems over robot operating system (ros) for finding correctness bugs. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 447–458, New York, NY, USA, 2022. Association for Computing Machinery.

[33] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing, 2018.

[34] Karsten Knese. rosbag2. `https://github.com/ros2/rosbag2`, 2019.

[35] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 562–566, 2018.

[36] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. 01 2005.

[37] Steve Macenski, Francisco Martín, Ruffin White, and Jonatan Ginés Clavero. The marathon 2: A navigation system. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[38] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[39] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[40] Federico Maggi, Rainer Vosseler, Mars Cheng, Patrick Kuo, Chizuru Toyama, Ta-Lun Yen, Erik Boasson, and Víctor Mayoral Vilches. A security analysis of the data distribution service (dds) protocol. Technical report, Trend Micro Research, August 2022.

[41] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10, 2016.

[42] Subash Neupane, Shaswata Mitra, Ivan Fernandez, Swayamjit Saha, Nisha Pillai, Jingdao Chen, Sudip Mittal, and Shahram Rahimi. Security considerations in ai-robotics: A survey of current methods, challenges, and opportunities, 10 2023.

[43] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six - mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275–378. Elsevier, 2019.

[44] Shariefuddin Pirzada. An introduction to graph theory. *Acta Universitatis Sapientiae*, 4(2):289, 2012.

[45] THE ASSOCIATED PRESS. Key events in the troubled history of the boeing 737 max. *THE ASSOCIATED PRESS*.

[46] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. volume 3, 01 2009.

[47] Robotis. Turtlebot3. `https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/`.

[48] Sergio Sandoval and Preetha Thulasiraman. Cyber security assessment of the robot operating system 2 for aerial networks. In *2019 IEEE International Systems Conference (SysCon)*, pages 1–8, 2019.

[49] Contrast Security. Fuzz testing. `https://www.contrastsecurity.com/glossary/fuzz-testing`.

[50] Kamal Taha. Semi-supervised and un-supervised clustering: A review and experimental evaluation. *Information Systems*, 114:102178, 2023.

[51] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero,

Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[52] Eric W Weisstein. Euler angles. *https://mathworld. wolfram. com/*, 2009.

[53] Alexander Wolf, August Carzaniga, David Rosenblum, and Er Wolf. Design of a scalable event notification service: Interface and architecture. 12 1998.

[54] Michael Wrona. quat2eulers.py. `https://gist.github.com/michaelwro/1450283a6a1226eaf707d9adde378798`, 2021.

[55] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Guide for authors. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023. Retrieved 2023-11-11 18:23:44+01:00.

Appendix A

Experiment Code

A.1 run_experiments.py

```
import subprocess
import time
import signal
import yaml
from pysdf import SDF
import random
import argparse
from graphs import graph_from_nodes


# If processes don't end:
# killall -INT /usr/bin/python3
def run(x, y, i, mode="lidar_passthrough"):
```

```
run_nav2_bringup = f"ros2 launch nav2_bringup
↪ tb3_simulation_launch.py headless:=False
↪ x_pose:={x} y_pose:={y}"
run_collision_monitor = "ros2 launch
↪ nav2_collision_monitor
↪ collision_monitor_node.launch.py"
run_nav_to_dest = "ros2 action send_goal
↪ /navigate_to_pose nav2_msgs/action/NavigateToPose
↪ \"pose: {header: {frame_id: map}, pose: {position:
↪ {x: -1.5, y: 1.5, z: 0.0}, orientation: {x: 0.0,
↪ y: 0.0, z: 0, w: 1.00000}}}\""
run_lidar_passthrough = "ros2 run lidar_passthrough
↪ lidar_passthrough"
run_lidar_passthrough_odom = "ros2 run
↪ lidar_passthrough_odom lidar_passthrough_odom"
run_fake_object = "ros2 run nav2 fakeobject"
run_random_noise = "ros2 run random_noise
↪ random_noise"


print("Starting nav2...")


if mode == "lidar_passthrough":
```

```
        f_passthrough =
        ↪  open(f"logs/lidar_passthrough_{i}.log", "w")
        f_passthrough_err =
        ↪  open(f"logs/lidar_passthrough_err_{i}.log",
        ↪  "w")
        print(run_lidar_passthrough)
        mode_cmd =
        ↪  subprocess.Popen(run_lidar_passthrough.split(),
        ↪  shell=False, stdout=f_passthrough,
        ↪  stderr=f_passthrough_err)

    elif mode == "lidar_passthrough_odom":
        f_passthrough_odom =
        ↪  open(f"logs/lidar_passthrough_odom_{i}.log",
        ↪  "w")
        f_passthrough_odom_err =
        ↪  open(f"logs/lidar_passthrough_odom_err_{i}.log",
        ↪  "w")
        print(run_lidar_passthrough_odom)
        mode_cmd =
        ↪  subprocess.Popen(run_lidar_passthrough_odom.split(),
        ↪  shell=False, stdout=f_passthrough_odom,
        ↪  stderr=f_passthrough_odom_err)
```

```
    elif mode == "fake_object":
        f_fake_object = open(f"logs/fake_object_{i}.log",
        ↪  "w")
        f_fake_object_err =
        ↪  open(f"logs/fake_object_err_{i}.log", "w")
        print(run_fake_object)
        mode_cmd =
        ↪  subprocess.Popen(run_fake_object.split(),
        ↪  shell=False, stdout=f_fake_object,
        ↪  stderr=f_fake_object_err)

    elif mode == "random_noise":
        f_random_noise =
        ↪  open(f"logs/random_noise_{i}.log", "w")
        f_random_noise_err =
        ↪  open(f"logs/random_noise_err_{i}.log", "w")
        print(run_random_noise)
        mode_cmd =
        ↪  subprocess.Popen(run_random_noise.split(),
        ↪  shell=False, stdout=f_random_noise,
        ↪  stderr=f_random_noise_err)


    f2 = open(f"logs/collision_monitor_run_{i}.log", "w")
```

```
f2_err =
↪  open(f"logs/collision_monitor_run_err_{i}.log",
↪  "w")
print(run_collision_monitor)
p2 = subprocess.Popen(run_collision_monitor.split(),
↪  shell=False, stdout=f2, stderr=f2_err)

f1 = open(f"logs/nav2_bringup_run_{i}.log", "w")
f1_err = open(f"logs/nav2_bringup_run_err_{i}.log",
↪  "w")
print(run_nav2_bringup)
print(f"logs in {f1.name} and {f1_err.name}")
p1 = subprocess.Popen(run_nav2_bringup.split(),
↪  shell=False, stdout=f1, stderr=f1_err)

print("Waiting for nav2_bringup")

print("writing pid to logs")
with open("logs/pid_nav2", 'w') as f:
    f.write(str(p1.pid)+"\n")
    f.write(str(p2.pid)+"\n")

print("Sleeping...")
time.sleep(30)
```

```
print(f"Recording bags...")


f3 = open(f"logs/ros_bag_record_{i}.log", "w")
f3_err = open(f"logs/ros_bag_record__err_{i}.log",
↪  "w")
bag_file_path = f"/media/labib/My
↪  Passport/experiment_bags/{mode}_run_{i}"
record_command_split = "ros2 bag record -a -o".split()
↪  + [bag_file_path]

record_proc = subprocess.Popen(record_command_split,
↪  shell=False, stdout=f3, stderr=f3_err)



if mode == "delayed_passthrough":
    print("Delayed Passthrough")
    f_passthrough_odom =
    ↪  open(f"logs/lidar_passthrough_odom_delayed_{i}.log",
    ↪  "w")
    f_passthrough_odom_err =
    ↪  open(f"logs/lidar_passthrough_odom_err_delayed_{i}.log",
    ↪  "w")
    print(run_lidar_passthrough)
```

```
        mode_cmd =
        ↪  subprocess.Popen(run_lidar_passthrough_odom.split(),
        ↪  shell=False, stdout=f_passthrough_odom,
        ↪  stderr=f_passthrough_odom_err)
        time.sleep(10)

    elif mode == "delayed_fakeobject":
        print("Delayed fakeobject")
        f_fake_object =
        ↪  open(f"logs/fake_object_delayed_{i}.log", "w")
        f_fake_object_err =
        ↪  open(f"logs/fake_object_err_delayed_{i}.log",
        ↪  "w")
        print(run_fake_object)
        mode_cmd =
        ↪  subprocess.Popen(run_fake_object.split(),
        ↪  shell=False, stdout=f_fake_object,
        ↪  stderr=f_fake_object_err)
        time.sleep(10)

    time.sleep(10)
    print(run_nav_to_dest)
    p3 = subprocess.Popen(run_nav_to_dest, shell=True,
    ↪  stdout=subprocess.PIPE, stderr=subprocess.PIPE)
```

```
    ↪ graph_from_nodes.create_graph(f"graphs/{mode}_run-{i}")

p3.wait(timeout=3600)

print("Killing processes...")
record_proc.send_signal(signal.SIGINT)
p1.send_signal(signal.SIGINT)
p2.send_signal(signal.SIGINT)
if mode == "lidar_passthrough":
    mode_cmd.send_signal(signal.SIGINT)
    f_passthrough.close()
    f_passthrough_err.close()
elif mode == "lidar_passthrough_odom" or mode ==
    ↪ "delayed_passthrough":
    mode_cmd.send_signal(signal.SIGINT)
    f_passthrough_odom.close()
    f_passthrough_odom_err.close()
elif mode == "fake_object" or mode ==
    ↪ "delayed_fakeobject":
    mode_cmd.send_signal(signal.SIGINT)
    f_fake_object.close()
    f_fake_object_err.close()
```

```
    elif mode == "random_noise":
        mode_cmd.send_signal(signal.SIGINT)
        f_random_noise.close()
        f_random_noise_err.close()

    f1.close()
    f1_err.close()
    f2.close()
    f2_err.close()
    f3.close()
    f3_err.close()
    i += 1

def update_params_yaml(yaml_file_path, x, y):

    # original pose: {'x': -2.0, 'y': -0.5, 'z': 0.0}
    with open(yaml_file_path) as f:
        params = yaml.safe_load(f)

    curr_pose =
    ↪  params["amcl"]["ros__parameters"]["initial_pose"]
    curr_pose['x'] = x
    curr_pose['y'] = y
    curr_pose['yaw'] = curr_pose['yaw']
```

```
    print(f"New pose: {curr_pose}")

    with open(yaml_file_path, "w") as f:
        yaml.dump(params, f)

def get_rand_pose():
    input_file = INPUT_FILE
    parsed = SDF.from_file(input_file)
    border = ["head","left_hand","right_hand",
    "left_foot","right_foot","body"]
    cylinders =
    ↪  ["one_one","one_two","one_three","two_one","two_two",
    "two_three","three_one","three_two","three_three"]
    border_pose_x = []
    border_pose_y = []

    cylinder_pose = []
    for collision in parsed.iter("collision"):
        (x,y,z,roll,pitch,yaw) = collision.pose.value
        if collision.name in border:
            border_pose_x.append(x)
            border_pose_y.append(y)
        elif collision.name in cylinders:
            cylinder_pose.append((x,y))
```

```
    while True:
        rand_x = random.uniform(min(border_pose_x)+1,
        ↪  max(border_pose_x)-1)
        rand_y = random.uniform(min(border_pose_y)+1,
        ↪  max(border_pose_y)-1)
        rand_pose = (rand_x, rand_y)
        print("Random pose: ", rand_pose)
        if rand_pose not in cylinder_pose:
            return rand_pose
        else:
            print("Random pose in cylinder")


if __name__ == "__main__":
    ALLOWED_MODES = ["lidar_passthrough",
    ↪  "lidar_passthrough_odom", "fake_object",
    ↪  "random_noise", "delayed_passthrough",
    ↪  "delayed_fakeobject"]
    parser = argparse.ArgumentParser()
    parser.add_argument("-m", "--mode", help="Experiment
    ↪  mode", choices=ALLOWED_MODES)
    parser.add_argument("-n", "--num_runs", help="Number
    ↪  of runs", type=int, default=1)
```

```
    args = parser.parse_args()



    for i in range(args.num_runs):
        rand_pose = get_rand_pose()
        update_params_yaml(PARAM_FILE, rand_pose[0],
        ↪  rand_pose[1])
        run(rand_pose[0], rand_pose[1], i, mode=args.mode)
        time.sleep(480)
```

A.2 Extract Vectors Code

A.2.1 config.yml

```
paths:
  root_dir: "/media/labib/My Passport/experiment_bags/"
  data_dir: "/media/labib/My Passport/"
  graph_dir: "/home/labib/IDS/Labib/tools/graphs"
  plugin_dir: "./plugins"


database:
  extension: "db3"
```

A.2.2 plugin_interface.py

```
    # plugin_interface.py
from abc import ABC, abstractmethod


class FeatureExtractionPlugin(ABC):
    @abstractmethod
    def extract_features(self, G, cur):
        pass
```

A.2.3 plugin_manager.py

```
# plugin_manager.py
import importlib.util
import os
from plugin_interface import FeatureExtractionPlugin


class PluginManager:
    def __init__(self, plugin_dir):
        self.plugin_dir = plugin_dir
        self.plugins = []


    def load_plugins(self):
        for filename in os.listdir(self.plugin_dir):
            if filename.endswith(".py"):
```

```
                module_name = filename[:-3]
                module_path =
                ↪  os.path.join(self.plugin_dir,
                ↪  filename)
                spec =
                ↪  importlib.util.spec_from_file_location(module_name,
                ↪  module_path)
                module =
                ↪  importlib.util.module_from_spec(spec)
                spec.loader.exec_module(module)
                for attr_name in dir(module):
                    attr = getattr(module, attr_name)
                    if isinstance(attr, type) and
                    ↪  issubclass(attr,
                    ↪  FeatureExtractionPlugin) and attr
                    ↪  is not FeatureExtractionPlugin:
                        self.plugins.append(attr())

    def extract_all_features(self, G, cur):
        features = []
        for plugin in self.plugins:
            print("Extracting: ", plugin)
            features.extend(plugin.extract_features(G,
            ↪  cur))
```

```
        return features
```

A.2.4 main.py

```
    # main.py
import os
import sqlite3
import pandas as pd
import networkx as nx
from plugin_manager import PluginManager
import yaml

# Read configurations
with open('config.yml', 'r') as file:
    config = yaml.safe_load(file)

ROOTDIR = config['paths']['root_dir']
DATADIR = config['paths']['data_dir']
GRAPHDIR = config['paths']['graph_dir']
PLUGINDIR = config['paths']['plugin_dir']
DB_EXTENSION = config['database']['extension']

def create_cursor(fn):
    cur, con = None, None
    try:
```

```
        con = sqlite3.connect(fn)
        cur = con.cursor()
    except sqlite3.Error as e:
        print(e)
    return cur, con

def convert_bag_name_to_graph(filename):
    idx_last_underscore = filename.rindex("_")
    updated_filename = filename[:idx_last_underscore]
    idx_last_underscore = updated_filename.rindex("_")
    final_filename =
    ↪  updated_filename[:idx_last_underscore] + "-" +
    ↪  updated_filename[idx_last_underscore+1:]
    return final_filename

def build_dataset():
    columns = ['filename', 'is_malicious', 'features']
    data = []

    plugin_manager = PluginManager(PLUGINDIR)
    plugin_manager.load_plugins()

    for subdir, dirs, files in os.walk(ROOTDIR):
        for file in files:
```

```
bag = os.path.join(subdir, file)
filename = file.split(".")[0]
print("File: ", filename)
if bag.endswith(DB_EXTENSION):
    cur, con = create_cursor(bag)
    graph_file = os.path.join(GRAPHDIR,
    ↪  convert_bag_name_to_graph(filename) +
    ↪  ".graphml")
    print(f"Reading graph from
    ↪  {graph_file}...")
    if not os.path.isfile(graph_file):
        print(f"{graph_file} not found")
        continue

    G = nx.read_graphml(graph_file)
    features =
    ↪  plugin_manager.extract_all_features(G,
    ↪  cur)
    print("Features: ", features)

    is_malicious = "fake" in filename or
    ↪  "random_noise" in filename
```

```
                row = {'filename': filename,
                ↪  'is_malicious': is_malicious,
                ↪  'features': features}
                data.append(row)

    df = pd.DataFrame(data)
    print(f"Writing to csv in {DATADIR}")
    df.to_csv(os.path.join(DATADIR, 'plugin_test.csv'))

def main():
    build_dataset()

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

A.2.5 contextual_features.py

```
# contextual_features.py
import numpy as np
from plugin_interface import FeatureExtractionPlugin
import math
from rclpy import serialization
import rosidl_runtime_py
from rosidl_runtime_py import utilities, convert
```

```
def quat2eulers(q0:float, q1:float, q2:float, q3:float) ->
↪  tuple:
    roll = math.atan2(
        2 * ((q2 * q3) + (q0 * q1)),
        q0**2 - q1**2 - q2**2 + q3**2
    )
    pitch = math.asin(2 * ((q1 * q3) - (q0 * q2)))
    yaw = math.atan2(
        2 * ((q1 * q2) + (q0 * q3)),
        q0**2 + q1**2 - q2**2 - q3**2
    )
    return (roll, pitch, yaw)

def deserialize(msg, msg_type, deserialize_form="dict"):
    # can convert to yaml, csv, or ordereddict
    #TODO: Catch error and record which run
    if deserialize_form == "yaml":
        return convert.message_to_yaml(
        serialization.deserialize_message(msg, msg_type))
    elif deserialize_form == "csv":
        return convert.message_to_csv(
        serialization.deserialize_message(msg, msg_type))
    return convert.message_to_ordereddict(
    serialization.deserialize_message(msg, msg_type))
```

```
class ContextualFeatureExtractor(FeatureExtractionPlugin):
    def extract_features(self, G, cur):
        yaws = []
        range_var = []
        contextual_fvs = []

        try:
            cur.execute("SELECT MIN(timestamp) FROM
            ↪ messages")
            min_time = int(cur.fetchone()[0])
            cur.execute("SELECT MAX(timestamp) FROM
            ↪ messages")
            max_time = int(cur.fetchone()[0])

            duration = max_time - min_time

            cur.execute("SELECT * FROM topics")
            topics = {row[0]: row for row in
            ↪ cur.fetchall()}
            cur.execute("SELECT * FROM messages")
            for msg in cur.fetchall():
                topic_info = topics[msg[1]]
                topic_name = topic_info[1]
```

```
serialized_msg = msg[3]
msg_type =
 ↪  utilities.get_message(topic_info[2])

if topic_name == '/odom':
    deserialized_msg =
     ↪  deserialize(serialized_msg,
     ↪  msg_type)
    orientation =
     ↪  deserialized_msg['pose']['pose']['orientation']
    roll, pitch, yaw =
     ↪  quat2eulers(orientation['w'],
     ↪  orientation['x'],
     ↪  orientation['y'],
     ↪  orientation['z'])
    yaws.append(yaw)

elif 'scan' in topic_name:
    deserialized_msg =
     ↪  deserialize(serialized_msg,
     ↪  msg_type)
    ranges = deserialized_msg['ranges']
    range_var.append(np.var(ranges))
```

```
            contextual_fvs.append(np.var(yaws))
            contextual_fvs.append(np.mean(range_var))
            contextual_fvs.append(duration)

        except Exception as e:
            print(e)

        return contextual_fvs
```

A.2.6 structural_features.py

```
# structural_features.py
import numpy as np
import networkx as nx
from plugin_interface import FeatureExtractionPlugin

class StructuralFeatureExtractor(FeatureExtractionPlugin):
    def extract_features(self, G, cur=None):
        degrees = dict(G.degree())
        degree_values = np.array(list(degrees.values()))
        degree_probabilities = degree_values /
        ↪  degree_values.sum()

        density = nx.density(G)
```

```
        entropy = -np.sum(degree_probabilities *
        ↪  np.log(degree_probabilities))
        degrees_hist = nx.degree_histogram(G)
        max_degree = degrees_hist.index(max(degrees_hist))
        avg_degree = sum(degrees_hist) / len(degrees_hist)
        variance = np.var(degree_values)

        in_degree_centrality =
        ↪  nx.algorithms.in_degree_centrality(G)
        max_centrality =
        ↪  max(in_degree_centrality.values())
        avg_centrality =
        ↪  sum(in_degree_centrality.values()) /
        ↪  len(in_degree_centrality)

        vectors = [max_degree, avg_degree, density,
        ↪  entropy, max_centrality, avg_centrality,
        ↪  variance]
        return vectors
```

A.3 cluster.py

```
from scipy.spatial.distance import pdist, squareform
from scipy.cluster.hierarchy import dendrogram, linkage,
↪  fcluster
```

```
from sklearn.metrics import roc_auc_score, roc_curve, auc,
↪  silhouette_score
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import ast
import json
import sys
sys.setrecursionlimit(10000)

DATASET = "/media/labib/My
↪  Passport/experiments_with_yaw_sensor_and_duration.csv"
EXPERIMENT_PREFIXES = ["lidar_passthrough",
↪  "lidar_passthrough_odom", "random_noise",
↪  "fake_object", "delayed_passthrough",
↪  "delayed_fakeobject"]

def convert_dataset_to_features(df, mode="all"):
    # Pad feature vectors to the same length
    features = []
    for feature in df['features']:
        if "nan" in feature:
            feature = feature.replace("nan", "0")
        # to list of int
```

```
            feature = json.loads(feature)
            if mode == "structural":
                features.append(feature[:7])
            elif mode == "contextual":
                features.append(feature[8:])
            else:
                features.append(feature)
        return np.array(features)

def bootstrap(df, n=1000, rate=0.2):

    is_malicious = df[df["is_malicious"] == True]
    non_malicious = df[df["is_malicious"] == False]

    n_malicious_to_keep = int(n * rate)

    non_malicious_jobs_sample =
    ↪  non_malicious.sample(n=(n-n_malicious_to_keep),
    ↪  random_state=50, replace=True)
    malicious_jobs_sample =
    ↪  is_malicious.sample(n=n_malicious_to_keep,
    ↪  random_state=51, replace=True)
```

```
    combined_df = pd.concat([non_malicious_jobs_sample,
    ↪  malicious_jobs_sample], ignore_index=True)
    return combined_df


def accuracy(df, t=2):
    total_malicious = len(df[df['is_malicious'] == True])
    max_acc = (None, 0)
    for i in range(t):
        num_rows = len(df[(df['cluster'] == i+1) &
        ↪  (df['is_malicious'] == True)])
        acc = num_rows/total_malicious
        print(acc, i+1)
        if acc > max_acc[1]:
            max_acc = (i+1, acc)
    accuracy = max_acc[1]
    cluster = max_acc[0]
    print(f"Accuracy: {accuracy} Cluster: {cluster}")
    print(f"AUC: {calculate_auc_roc(df, cluster)}")
    plot_roc_curve(df, cluster)
    return accuracy


def calculate_auc_roc(df, target_cluster):
    # Create a binary label based on whether the sample is
    ↪  in the target cluster
```

```
    df['binary_label'] = df['cluster'] == target_cluster

    # Calculate AUC-ROC
    auc_value = roc_auc_score(df['is_malicious'],
    ↪  df['binary_label'])
    return auc_value

def plot(features, clusters,
↪  fig_name="clustering_figure.png"):
    # Plot clustering result
    plt.figure(figsize=(8, 6))
    plt.scatter(features[:, 2], features[:, 1],
    ↪  c=clusters, cmap='viridis')
    plt.title('Agglomerative Hierarchical Clustering')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.colorbar(label='Cluster')
    plt.grid(True)
    plt.tight_layout()

    # Save the clustering figure
    plt.savefig(fig_name)

    # Show the clustering figure
```

```
    plt.show()


def plot_roc_curve(df, target_cluster):
    # Create a binary label based on whether the sample is
    ↪  in the target cluster
    df['binary_label'] = df['cluster'] == target_cluster


    # Compute ROC curve and AUC
    fpr, tpr, _ = roc_curve(df['is_malicious'],
    ↪  df['binary_label'])
    roc_auc = auc(fpr, tpr)
    print(f"FPR: {fpr}, TPR: {tpr}, ROC_AUC: {roc_auc}")
    # Plot ROC curve
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='darkorange', lw=2,
    ↪  label=f'ROC curve (area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2,
    ↪  linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic')
    plt.legend(loc="lower right")
```

```
    plt.grid(True)
    plt.tight_layout()
    plt.savefig("roc.png")

def grid_search(combined_df, features, methods=['ward',
↪  'single', 'complete', 'average'],
↪  criteria=['maxclust', 'distance']):
    best_score = -1
    best_params = None
    best_clusters = None

    distances = pdist(features)
    for method in methods:
        Z = linkage(distances, method=method)
        for criterion in criteria:
            print(f"Method: {method} | Criterion:
            ↪  {criterion}")
            plt.figure(figsize=(10, 7))
            dendrogram(Z)

            ↪  plt.savefig(f"{method}-{criterion}-dendogram.png")
            t = 2
            if method == "ward":
```

```
            clusters = fcluster(Z, t=2,
            ↪ criterion=criterion)
        else:
            t = 4
            clusters = fcluster(Z, t=4,
            ↪ criterion=criterion)

        # score = silhouette_score(features, clusters)
        print(f"Max Cluster: {max(clusters)} Min
        ↪ Cluster: {min(clusters)}")
        # print(f"Method: {method}, Criterion:
        ↪ {criterion}, Silhouette Score:
        ↪ {score:.4f}")
        combined_df['cluster'] = clusters

        acc = accuracy(combined_df, t)
        if acc > best_score:
            best_score = acc
            best_params = {'method': method,
            ↪ 'criterion': criterion}
            best_clusters = clusters

print(f"\nBest Parameters: {best_params}, Best Acc:
↪ {best_score:.4f}")
```

```
    return best_params, best_score, best_clusters


def subset_dataset(df, prefixes):
    appended_prefixes = []
    for prefix in prefixes:
        if prefix not in EXPERIMENT_PREFIXES:
            print(f"Prefix {prefix} not in experiments")
            return df
        else:
            appended_prefixes.append(prefix+"_run")

    mask =
    ↪  df['filename'].str.startswith(tuple(appended_prefixes))
    filtered_df = df[mask]
    return filtered_df


def main():
    df = pd.read_csv(DATASET)
    df = df.fillna(0)
    df = subset_dataset(df, ["delayed_passthrough",
    ↪  "delayed_fakeobject"])

    # for i in [0.1, 0.3, 0.5, 0.7, 0.9]:
    combined_df = bootstrap(df, 10000, 0.5)
```

```
mode = "delayed_lidar_odom_fake_object_contextual"
features = convert_dataset_to_features(combined_df,
↪  mode="contextual")
distances = pdist(features)

# grid_search(combined_df, features)

# # Perform hierarchical clustering
Z = linkage(distances, method='ward')
plt.figure(figsize=(10, 7))
dendrogram(Z)
plt.savefig(f"{mode}-ward-maxclust-dendogram.png")
clusters = fcluster(Z, t=2, criterion='maxclust')

# params, score, clusters = grid_search(combined_df,
↪  features) # => {'method': 'ward', 'criterion':
↪  'maxclust'}
combined_df['cluster'] = clusters
# print(combined_df[['filename', 'cluster']])

accuracy(combined_df, 2)

plot(features, clusters)
```

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

A.4 ROS2 Nodes

A.4.1 random_noise.py

```
    import rclpy
from rclpy.node import Node
import random

from sensor_msgs.msg import LaserScan


class LidarPassthroughWithNoisePublisher(Node):

    def __init__(self, sub_topic, pub_topic):
        super().__init__('lidar_passthrough')
        self.publisher_ = self.create_publisher(LaserScan,
        ↪  pub_topic, 10)
        self.subscriber =
        ↪  self.create_subscription(LaserScan, sub_topic,
        ↪  self.lidar_callback, 10)

    def lidar_callback(self, laserscan_msg):
```

```
        noisy_laserscan_msg = LaserScan()
        noisy_laserscan_msg.header = laserscan_msg.header
        noisy_laserscan_msg.angle_min =
        ↪  laserscan_msg.angle_min
        noisy_laserscan_msg.angle_max =
        ↪  laserscan_msg.angle_max
        noisy_laserscan_msg.angle_increment =
        ↪  laserscan_msg.angle_increment
        noisy_laserscan_msg.time_increment =
        ↪  laserscan_msg.time_increment
        noisy_laserscan_msg.scan_time =
        ↪  laserscan_msg.scan_time
        noisy_laserscan_msg.range_min =
        ↪  laserscan_msg.range_min
        noisy_laserscan_msg.range_max =
        ↪  laserscan_msg.range_max

        # Add random noise to each range value
        noisy_laserscan_msg.ranges = [
            self.add_random_noise(range_value) for
            ↪  range_value in laserscan_msg.ranges
        ]

        # Copy intensities if available
```

```
        if laserscan_msg.intensities:
            noisy_laserscan_msg.intensities =
            ↪ laserscan_msg.intensities

        self.publisher_.publish(noisy_laserscan_msg)

    def add_random_noise(self, value, noise_level=0.1):
        noise = random.uniform(-noise_level * value,
        ↪ noise_level * value)
        return value + noise



def main(args=None):
    rclpy.init(args=args)

    lidar_passthrough =
    ↪ LidarPassthroughWithNoisePublisher("/scan_tom",
    ↪ "/scan_fakeobject")

    rclpy.spin(lidar_passthrough)

    lidar_passthrough.destroy_node()
    rclpy.shutdown()
```

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

A.4.2 lidar_passthrough.py

```
import rclpy
from rclpy.node import Node

from sensor_msgs.msg import LaserScan


class LidarPassthroughPublisher(Node):

    def __init__(self, sub_topic, pub_topic):
        super().__init__('lidar_passthrough')
        self.publisher_ = self.create_publisher(LaserScan,
        ↪  pub_topic, 10)
        self.subscriber =
        ↪  self.create_subscription(LaserScan, sub_topic,
        ↪  self.lidar_callback, 10)


    def lidar_callback(self, msg):
```

```
        self.publisher_.publish(msg)



def main(args=None):
    rclpy.init(args=args)

    lidar_passthrough =
    ↪ LidarPassthroughPublisher("/scan_tom",
    ↪ "/scan_fakeobject")

    rclpy.spin(lidar_passthrough)

    lidar_passthrough.destroy_node()
    rclpy.shutdown()



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

A.4.3 lidar_passthrough_odom.py

```
    import rclpy
from rclpy.node import Node
```

```
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import Odometry

class LidarPassthroughOdomPublisher(Node):

    def __init__(self, sub_topic, pub_topic):
        super().__init__('lidar_passthrough')
        self.publisher_ = self.create_publisher(LaserScan,
        ↪  pub_topic, 10)
        self.subscriber =
        ↪  self.create_subscription(LaserScan, sub_topic,
        ↪  self.lidar_callback, 10)
        self.subscriber_odom =
        ↪  self.create_subscription(Odometry, "/odom",
        ↪  self.lidar_callback_odom, 10)


    def lidar_callback(self, msg):
        self.publisher_.publish(msg)

    def lidar_callback_odom(self, msg):
        # do nothing
        print(msg)
```

```
def main(args=None):
    rclpy.init(args=args)

    lidar_passthrough =
    ↪ LidarPassthroughOdomPublisher("/scan_tom",
    ↪ "/scan_fakeobject")

    rclpy.spin(lidar_passthrough)

    lidar_passthrough.destroy_node()
    rclpy.shutdown()


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