PhD-FSTM-2021-015
The Faculty of Sciences, Technology and Medicine

DISSERTATION

Defence held on 19/01/2021 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Sean Arthur RIVERA
Born on 15 June 1992 in Colorado Springs, CO, United States of America

SECURING ROBOTS: AN INTEGRATED APPROACH FOR SECURITY CHALLENGES AND MONITORING FOR THE ROBOTIC OPERATING SYSTEM

Dissertation defence committee

Dr Radu STATE, dissertation supervisor
Professor, Université du Luxembourg

Dr Vijay GURBANI
Illinois Institute of Technology

Dr Gabriele LENZINI, Chairman
Professor, Université du Luxembourg

Dr Sheila BECKER
Institut Luxembourgeois De Régulation

Dr Cristina NITA-ROTARU, Vice Chair
Professor, Northeasten University

Affidavit

I hereby confirm that the PhD thesis entitled “Securing Robots: An Integrated Approach for Security Challenges and Monitoring for the Robotic Operating System (ROS)” has been written independently and without any other sources than cited.

Luxembourg, ____________________ ______________________________

Name

Acknowledgments

I want to thank the Interdisciplinary Centre for Security Reliability and Trust (SnT) at the University of Luxembourg for their assistance with my studies. I would also like to thank the Luxembourg Directorate of Defense and the CONCORDIA project for all of their support with my research. Without all of them, I would have been unable to pursue my Ph.D.

I would like to thank my supervisor Radu State, for inviting me out to Luxembourg for my Ph.D. and for all of his guidance and support during my studies. I really appreciated all of the flexibility in allowing me to explore this topic and the availability of support.

I would like to extend my thanks to my CET members, Radu State, Sheila Becker, and Sofiane Lagraa, for all the time they spent helping me. Our meetings were always helpful as both guideposts for my research.

My sincerest thanks are extended to my defense jury members, Gabriele Lenzini , Radu State, Sheila Becker, Cristina Nita-Rotaru, Vijay Gurbani, Sofiane Lagraa, and Antonio Ken Iannillo for all their help with my dissertation and all the time they spent participating in my defense. I would like to further thank Christina Nita-Rotaru and Vijay Gurbani for their assistance with my research throughout my Ph.D.

I would like to thank my colleagues and co-workers in SEDAN labs for all the fun times we've shared and the ideas we've bounced off one another. I would like to extend my thanks to Antonio Ken Iannillo and Sofiane Lagraa for all their assistance in our research collaborations.

Last I want to thank my fiancee Mary Roszel, for all of her support and all the late nights spent working on papers together and my family for all of their love and support.

Dedications

I dedicate this dissertation to my fiancee Mary Roszel, for all the time and effort she put into helping me. Without her, so much would have been left undone that I would not have been able to finish this dissertation. I love you, Mary.

Abstract

Robotic systems are proliferating in our society due to their capacity to carry out physical tasks on behalf of human beings, with current applications in the military, industrial, agricultural, and domestic fields. The Robotic Operating System (ROS) is the de-facto standard for the development of modular robotic systems. Manufacturing and other industries use ROS for their robots, while larger companies such as Windows and Amazon have shown interest in supporting it, with ROS systems projected to make up most robotic systems within the next five years. However, a focus on security is needed as ROS is notorious for the absence of security mechanisms, placing people in danger both physically and digitally.

This dissertation presents the security shortcomings in ROS and addresses them by developing a modular, secure framework for ROS. The research focuses on three features: internal system defense, external system verification, and automated vulnerability detection. This dissertation provides an integrated approach for the security of ROS-enabled robotic systems to set a baseline for the continual development of ROS security.

Internal system defense focuses on defending ROS nodes from attacks and ensuring system safety in compromise. ROS-Defender, a firewall for ROS leveraging Software Defined Networking (SDN), and ROS-FM, an extension to ROS-Defender that uses the extended Berkely Packet Filter(eBPF), are discussed. External system verification centers on when data becomes the enemy, encompassing sensor attacks, network infrastructure attacks, and inter-system attacks. In this section, the use of machine learning to address sensor attacks is demonstrated, eBPF is utilized to address network infrastructure attacks, and consensus algorithms are leveraged to mitigate inter-system attacks. Automated vulnerability detection is perhaps the most important, focusing on detecting vulnerabilities and providing immediate mitigating solutions to avoid downtime or system failure. Here, ROSploit, an automated vulnerability scanner for ROS, and DiscoFuzzer, a fuzzing system designed for robots, are discussed. ROS-Immunity combines all the components for an integrated tool that, in conjunction with Secure-ROS, provides a suite of defenses for ROS systems against malicious attackers.

Index

1 Introduction **2**

1.1 Thesis Structure 6

1.2 Contributions 8

1.2.1 As First Author: 8

1.2.2 As Co-Author: 12

2 Background **14**

2.1 Robotics and Robotic Operating Systems 14

2.1.1 Robotics 14

2.1.2 Robotic Operating Systems 16

3 State of the Art **25**

3.1 General Robotic Security 25

3.1.1 Common Robotic Systems 26

3.1.2 Robotic Components and Vulnerabilities 28

3.2 ROS Security 29

3.2.1 Security of the ROS network stack 31

3.2.2 Vulnerabilities in ROS 34

I Integrated Security **36**

4 Integrated ROS Security **37**

4.1 Introduction . . . 37
4.2 ROS System Designs: Centralized and Decentralized Systems . . . 40
4.3 Overall Architecture . . . 42
4.3.1 Threat Model . . . 42
4.3.2 Solution . . . 44
4.4 Robustness Assessment . . . 45
4.5 Automatic rule generation & Distribution . . . 47
4.6 Distributed Firewall . . . 50
4.7 Use Cases & Experimental Evaluation . . . 54
4.8 Conclusion and future work . . . 63

II Internal system defense **65**

5 ROSDefender **66**
5.1 Introduction . . . 66
5.2 Adversarial Model . . . 68
5.2.1 Independent Assumptions . . . 68
5.2.2 Attacker Model for ROS Deployment . . . 68
5.2.3 Attacker Model for SROS Deployment . . . 69
5.3 ROS-Defender Design . . . 69
5.3.1 Design Goals and Overview . . . 70
5.3.2 ROSDN . . . 72
5.3.3 ROSWatch . . . 74
5.3.4 ROS-Policy-Language . . . 75
5.3.5 Implementation . . . 76
5.4 Experimental Results . . . 76
5.4.1 Experimental Setup . . . 76
5.4.2 ROSDN Evaluation . . . 77
5.5 Conclusion and future work . . . 79

5.6 Key Contributions to ROS-Immunity 80

6 ROS-FM 81

6.1 Introduction 81
6.2 ROS monitoring tool 82
6.2.1 Design Goals and Overview 82
6.2.2 Architecture Overview 84
6.2.3 Domain-Specific Language 84
6.2.4 eBPF/XDP for ROS 87
6.2.5 Visualization Module 89
6.2.6 Implementation 90
6.3 Experiments 90
6.3.1 Experimental Setup 91
6.3.2 Overhead analysis 92
6.3.3 Security Evaluation 93
6.3.4 Monitoring Evaluation 95
6.3.5 Comparison and discussion 95
6.4 Conclusion and future work 95
6.5 Key Contributions to ROS-Immunity 96

7 DNS Privacy: A Use-Case for *ROS-FM* 97

7.1 Threat Model 98
7.1.1 Attack Descriptions 100
7.2 Methodology 101
7.2.1 Design Goals and Overview 101
7.2.2 Implementation 101
7.2.3 eBPF for DNS 104
7.2.4 eBPF for DoT 105
7.2.5 eBPF for DoH 105
7.3 Experiments & Results 106

7.3.1 Performance Experiment Setup 106
7.3.2 Performance Experiment Results 108
7.3.3 Privacy Experiment Scenarios 108
7.3.4 Privacy Experiment Setup 110
7.3.5 Privacy Experiment Results 111
7.3.6 Discussion 112
7.4 Conclusion and future work 115
7.4.1 Conclusion 115
7.4.2 Future Work 115

III External System Verification **116**

8 State Encoding **117**
8.1 Introduction 117
8.2 Spoofing Attacks Model 118
8.2.1 Independent Assumptions 118
8.2.2 Potential Spoofing Attacks 119
8.3 Architecture & Approach 122
8.3.1 Architecture 122
8.3.2 Learning Phase 125
8.3.3 Detection Phase 126
8.4 Experimental Results 126
8.4.1 Setup 126
8.4.2 Dataset Collection 127
8.4.3 Model Tuning 127
8.4.4 Experimental Results 128
8.5 Conclusion 130
8.6 Key Contributions to ROS-Immunity 131

IV Automated Vulnerability Detection **132**

9 ROSploit **133**

9.1 Introduction 133
9.2 ROS Framework 134
9.2.1 Threat Model 134
9.2.2 ROS attacks 136
9.2.3 ROSploit 137
9.2.4 Experimental Evaluation 142
9.3 Conclusion and future works 144
9.4 Key Contributions to ROS-Immunity 145

10 Discofuzzer **146**

10.1 Introduction 146
10.2 Discontinuity-Based Fuzz Testing 148
10.2.1 Sampling approaches 150
10.2.2 Discontinuity Analysis approach 151
10.3 DiscoFuzzer 153
10.3.1 Test Case Generator and Publisher 153
10.3.2 Output analyzer and Subscriber 156
10.3.3 Crash Detection 158
10.4 Evaluation 160
10.4.1 Benchmarks 160
10.4.2 Experimental design 162
10.4.3 Effectiveness of DiscoFuzzer 163
10.4.4 Efficiency of the sampling approaches 165
10.4.5 Discussion 168
10.4.6 Threat to validity 171
10.5 Conclusion 173
10.6 Key Contributions to ROS-Immunity 173

11 Discussion and Conclusions 174

11.0.1 Future Work 177

11.0.2 Closing Remarks 177

List of Figures

2.1 The Universal Robot's node and topics 19
2.2 ROS Architecture 20
2.3 Example of a Simple ROS System 22
2.4 ROS Centralized System Design 23
2.5 ROS Decentralized System Design 23

4.1 *ROS-Immunity*'s three components to implement an integrated security mechanism in ROS system addressing the security gaps in current ROS systems. 44
4.2 Integrated Firewall Architecture for centralized and decentralized ROS system designs. 52
4.3 *ROS-Immunity*: Results of Experimentation on Self-Driving Car 57
4.4 *ROS-Immunity*: Results of Experimentation on Swarm 58
4.5 *ROS-Immunity*: Results of Experimentation on Centralized Factory 59
4.6 *ROS-Immunity*: Results of Experimentation on Decentralized Factory 59
4.7 *ROS-Immunity*: Results of Experimentation: Normalized Power Overhead . . 63

5.1 ROS-Defender Design 70

6.1 *ROS-FM*: Visual representation of monitoring system 87
6.2 Analysis of ROS1 tools vs *ROS-FM* 91
6.3 *ROS-FM*: Node tracking example 92

7.1 Normal functionality of DNS, DoT, and DoH vs. eBPF implementation 102

7.2 *ROS-FM* and eBPF: Experimental Network overview 107
7.3 DNS lookup time measures of the Alexa top 50 109

8.1 Architecture of the encoder and decoder. 123
8.2 State Encoding Reconstruction Error 129

9.1 A graphical representation of the threat model. 135
9.2 *ROSploit* Evil Twin Diagram 136
9.3 *ROSploit* MiTM Diagram 137
9.4 *ROSploit* Example Scan 143
9.5 Turtlebot3 Reference robot. 144

10.1 Overview of DiscoFuzzer 159
10.2 *DiscoFuzzer* metrics comparison - detector mechanism 164
10.3 *DiscoFuzzer* metrics comparison - sampling approach 166
10.4 Venn diagram of the discovered *DiscoFuzzer* vulnerabilities 167
10.5 *DiscoFuzzer* Detection Times 169

11.1 The development pathway for *ROS-Immunity*. 175

List of Tables

3.1 Vulnerabilities in ROS Middleware 34
3.2 Other ROS vulnerabilities 35

4.1 Strengths and Weaknesses of known ROS security tools 46
4.2 Breakdown of the use-cases and experimental evaluation of *ROS-Immunity* . 56
4.3 *ROS-Immunity*: False positive rate per trial 58

5.1 ROSDN: ROS Turtlebot Experimental Results 78
5.2 ROSDN: ROS Virtual Result comparisons 79

6.1 *ROS-FM*: Effectiveness of *ROS-FM*on ROSPenTO 94
6.2 *ROS-FM*: Effectiveness of *ROS-FM*on AWS ROS2 SEC Test Node 94

7.1 Evaluation of security measures for DoT and DoH (with and without *ROS-FM*) 114

8.1 Detection Latency in Number of Samples (*ND*: not detected) 128

9.1 ROS DOS attacks 138
9.2 ROS data attacks 139
9.3 ROS node attacks 140

10.1 *DiscoFuzzer* Benchmarks and Vulnerability Sources 161
10.2 *DiscoFuzzer*'s configuration parameters used in the evaluation. 162
10.3 *DiscoFuzzer* statistical analysis 165
10.4 *DiscoFuzzer* Metrics 172

10.5 Pairwise comparison of the various *DiscoFuzzer* sampling approaches 172

Chapter 1

Introduction

"The ending is nearer than you think, and it is already written. All that we have left to choose is the correct moment to begin."

Alan Moore

Robotic systems are proliferating in our society due to their capacity to carry out physical tasks on behalf of human beings. The use of robotic systems began with small manufacturing applications, but the field of robotics has grown to include applications in every field with profound impact. Current applications include, but are not limited to, medical, military, industrial, agricultural, and domestic robots [95]. The role of robotics in society has continuously increased, spanning from industrial robots, consumer robots, commercial robots, vehicles, and drones. In western society, individuals interact with several robotic systems every day, receiving deliveries from companies utilizing robotic sorting machines, purchasing groceries packaged using robotic systems, and driving in cars built by robotic systems, running robotic software. The robotics market is ever-growing with a value of $115B in 2019 [196] and an expected CAGR of 25%. The role of robotics in society is of great importance, and close attention to detail is warranted.

Robotic systems are usually larger systems of interconnected, distributed components. Robots are cyber-physical systems designed with both physical and digital components. With robots, the physical world is highly coupled with cyberspace. Robots are characterized

by a certain degree of movement and autonomy to perform tasks [208], consisting of software that controls both a sensor and a mechanical part, acting upon the processed sensor data. Firstly, sensors perceive the physical environment; then, the control software chooses actions, potentially in collaboration with other agents in cyberspace (*e.g.*, other robots or the cloud). Finally, actuators perform those actions in the physical environment. Due to the complexity of robotic systems and their applications, most scenarios force the design of such systems to require that more than one robot is physically distributed in the environment. Often, there is a need to communicate between robots over a network to perform a common task.

For example, an everyday use case of robotics occurs in automobile manufacturing, where large robotic arms construct a car based on predetermined specifications. These systems require the coordination of many individual robots for planning and processing, including a centralized system where an operator provides instructions. In a single factory, tens to hundreds of robots may be working together for one goal. Even beyond a single factory, robots network over the internet to perform ever-greater feats of coordination, such as the Meggitt M4[121] project where several factories coordinated together to adapt assembly based on downstream needs.

Robotic systems require dedicated software to observe and understand the physical world, make decisions, plan and execute actions, and network. Many robotic operating systems are developed to coordinate robot-specific software. Of these systems, the most popular and widespread is The Robotic Operating System. The Robotic Operating System (ROS) is a framework for robotic system development, which is popular in both academic and industrial contexts. ROS is used in a multitude of real world settings, including industrial [85], consumer and commercial [102], self-driving cars [152], and military [182].

ROS is a collection of software libraries that enables communication of both (abstracted) hardware and (pure) software components to develop robotic systems [15]. In the last ten years, ROS became a de-facto standard for industrial applications powered by the ROS-Industrial[172], a consortium of companies worldwide that extends the advanced capabilities of ROS software to industrial relevant hardware and applications. It is predicted that

ROS centered systems will make up the majority of all robotic systems within the next five years, both in commercial and academic settings[149]. Of particular mention is the ROS-Industrial[172] consortium: it consists of 78 international industries that chose to extend and adopt ROS for their industrial robots.

ROS is an open-source project, and it has a vibrant community with thousands of developers and over nine thousand unique packages available[1]. Usually, these packages extend ROS core functionalities by implementing commonly used robotic software modules such as hardware drivers, robot models, datatypes, planning, perception, localization, simulation tools, and other algorithms. These robotic software modules are based on a modular design and, thus, highly re-usable, allowing for rapid and straightforward robotic system design.

ROS enables the development of robotic systems at a fine-grained scale. The design paradigm behind ROS is that of a Publish-Subscribe model, where a master keeps track of the state of the system while applications called nodes directly interact with each other through *topics*. A node is a process that performs a specific computation. A **ROS node** is a self-contained process that controls a part of the robot's operation. There can be a node computing a trajectory, a node moving the wheels, a node controlling a camera, *etc.*. A robotic system usually consists of many nodes that communicate with each other by passing messages. Messages define clean and consistent interfaces. A **ROS topic** is an implementation of a channel in which messages are passed in a publish-subscribe model. Topics act as a named bus where nodes can join as either a publisher, a subscriber, or both. When a publisher node sends a message over a topic, every subscriber node receives a copy. A particular node, namely the master node, is always present in a ROS-enabled system.

In the deployment of robots, it is vital to consider their safety-critical nature. As robotic systems become standard throughout industry and research, there is growing concern about potential security vulnerabilities. A hacked robot used in a public space like a supermarket, self-driving car, or private home can have severe consequences for humans' safety, especially as networking allows attackers to cause a cyberattack with physical impact remotely. ROS is notorious for the absence of security mechanisms, only partially covered by recent

[1] https://metrics.ros.org/

advancements with ROS2[193]. As ROS was not developed with industrial use in mind, it does not have any built-in security features for its base implementation. Although ROS is widely implemented, there are significant security vulnerabilities not addressed within its core framework. Several existing works related to robotics security can be found in the literature. Recent work highlighted several security threats against ROS [79, 64, 70]. The lack of proper security tools in ROS exposes systems to a variety of security concerns, including but not limited to sensor spoofing(the action of disguising a communication from an unknown source as being from a known one [77]), node impersonation, denial of service, man-in-the-middle, and privilege escalation attacks. The Open Robotics Foundation, creators of ROS, acknowledges the security concerns of ROS and ROS2.

For example, robots targeted by sensor spoofing attacks can force an incorrect behavior in the robotic system and undermine the success and safety of critical operations[55], potentially causing physical accidents in factories or other physical spaces. An insecure robot can irreversibly damage the physical environment in which it operates, including being harmful to human beings. These security flaws threaten the safety of systems built using the ROS framework, thus theoretically endangering users and related systems. Due to ROS's widespread use in large and significant industries, it is a significant and vulnerable target for malicious attacks.

Recent efforts have been made to add security features to ROS such as TLS and DTLS [68] for secure communication between nodes, web tokens for achieving secure authentication for remote access [40], and cryptographic methods that ensure data confidentiality and integrity [72]. The ROS group has also begun work on SROS [64], a security suite for ROS systems, still highly experimental and not fully implemented into the core ROS framework. In SROS, all network communication is encrypted using Secure Sockets Layer (SSL), specifically TLS. The encryption is done through the use of Public Key Infrastructure (PKI), where each ROS node is provided an x.509 certificate, equivalently an asymmetric key pair, signed by a trusted certificate authority. These results indicate security vulnerabilities in ROS, requiring additional libraries to ensure security in vital ROS systems.

The release of ROS2 addressed some security issues, but many security vulnerabilities

remain. Notably, the DDS security standard was introduced to provide secure communication between systems. However, this does not address other security vulnerabilities and leaves ROS systems open to various other attacks. For example, current solutions for the security of ROS and ROS2 do not provide protection against compromised nodes or denial of service, do not support reactive policies that are updated dynamically at run-time, and cannot enforce low-level granularity network policies.

To effectively monitor such robotic systems at run-time, special monitoring software is required. However, not many concepts and solutions have been proposed to monitor robotic systems' components, especially in ROS. Security in ROS is an active concern, with few fully implemented solutions [56]. Any such monitoring systems have to be flexible, scalable, and secure without sacrificing run-time. Most monitoring software is currently processing intensive and slow, a double disadvantage on cyber-physical systems, particularly robotic systems. There are no current solutions that offer both monitoring and security without a severe processing impact. A consistent set of security tools is crucial for security research as it provides a foundation for further development. Additionally, such tools allow a developer to test their systems against similar real-world attacks and more effectively develop secure reliable systems.

1.1 Thesis Structure

In this thesis, robotic security vulnerabilities, especially in ROS systems, are discussed, and several novel solutions are demonstrated. A fully integrated solution is demonstrated first, and then it is broken down into each of its components. Each of these components will discuss the solution's strengths and weaknesses and what they contribute to the total approach. This thesis is then structured into three broad pillars of research that made up the integrated solution's foundation. Those pillars are Internal System Defense (Part II), External System Verification (Part III), and Automated Vulnerability Detection (Part IV).

Chapter 2 provides in-depth background information on robotic operating systems, security vulnerabilities, and technologies to address these vulnerabilities. The Robotic Operating System, ROS, is introduced, as well as common security vulnerabilities that impact such sys-

tems. In addition, this chapter provides information on the technologies utilized throughout the remaining chapters and includes relevant information on topics such as autoencoders, Domain Name System (DNS), extended Berkeley Packet Filter (eBPF), software-defined networking (SDN), and fuzzing techniques.

Chapter 3 presents relevant work and discusses the current state of the art in robotics security research. A review of current approaches for security vulnerabilities is conducted to provide context on the solution presented in this thesis.

Chapter 4 outlines the integrated solution that is the culmination of this thesis, ROS-Immunity. An integrated packet-filtering firewall, automatic vulnerability detection system is discussed as the foundation of the thesis. The components of this system will be discussed in the following chapters in more depth.

Chapter 5 includes the preliminary foundation for the integrated system's firewall component. ROS-Defender is introduced, an SDN based firewall tool for ROS1 that addresses a variety of security vulnerabilities, including denial of service, man-in-the-middle, compromised ROS node attacks. The strengths and weaknesses of this approach are discussed. ROS-Defender provided the framework for the integrated system that will be used to grow upon in subsequent chapters.

Chapter 6 introduced ROS-FM, an updated implementation of ROS-Defender. ROS-FM includes necessary changes to ROS-Defender in speed, customization, and ease-of-use by replacing SDN with eBPF and implementing a full language specification in place of simple control commands. In addition, ROS-FM introduced advanced monitoring tools and the ability for ROS engineers to design custom plug-ins. ROS-FM is a core foundation to the firewall of ROS-Immunity.

The next two chapters discuss security vulnerabilities that could not be fully addressed using the firewall. Specific attention is given to sensor and user-privacy attacks. These attacks were focused on as very little research addressed them. The immense challenge of sensor attacks is that they require knowledge of normal behavior and very efficient monitoring to detect them before they damage the robot.

Chapter 8 discusses the use of autoencoders to detect sensor attacks, explicitly focusing

on LiDAR-based attacks. Sensor attacks are addressed due to their importance in robotics and potential impacts, as they have a significant role in robot functioning and safety protocols. This tool takes advantage of the contiguous nature of the real world to discuss discrepancies.

Chapter 7 provides the framework of a tool utilizing eBPF to address user-privacy attacks from DNS providers. This tool dynamically changes DNS providers to minimize DNS leakage between user queries, increasing user-privacy during normal internet usage. Demonstration of the framework is conducted with an analysis of the tool's success in blocking attacks compared to commonly utilized tools. This framework utilizes the tools built for ROS-FM and demonstrates the versatility of ROS-FM plug-ins.

The final two chapters discuss the automated vulnerability detection components of ROS-Immunity, intending to improve vulnerability detection speed, and reduce user intervention. Chapter 9 introduces an automated vulnerability scanner for ROS, ROSploit. This tool ensures that all security systems are correctly set up by checking for known vulnerabilities and simulating attacks. ROSploit is the primary tool used to check for *known* security vulnerabilities.

Chapter 10 introduces *DiscoFuzzer*, an automated vulnerability detection tool that addresses *unknown* security vulnerabilities by developing robot-specific fuzzing technologies. *DiscoFuzzer* forms the backbone for novel vulnerability detection for ROS. The tool is outlined as the primary fuzzing implementation used in ROS-Immunity.

This thesis is concluded in Chapter 11 where this thesis's contributions are summarized. Key contributions are presented.

1.2 Contributions

In this thesis, a variety of contributions are discussed. The following contributions were completed. Contributions are listed in chronological order.

1.2.1 As First Author:

- Rivera, Sean, Lagraa, Sofiane, and State, Radu. ROSploit: Cybersecurity Tool for ROS. In: *3rd IEEE International Conference on Robotic Computing*, IRC 2019, Naples,Italy,

February 25-27, 2019. 2019, pp. 415–416. [153] (*Included in Chapter 9*)

Robotic Operating System(ROS) security research is currently in a preliminary state, with limited research in tools or models. Considering the trend of digitization of robotic systems, this lack of foundational knowledge increases the potential threat posed by security vulnerabilities in ROS. This work presents a new tool to assist further security research in ROS, *ROSploit*. *ROSploit*is a modular two-pronged offensive tool covering both reconnaissance and exploitation of ROS systems, designed to assist researchers in testing exploits for ROS.

- Rivera, Sean et al. Ros-defender: SDN-based security policy enforcement for robotic applications. In: *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2019,pp. 114–119. [155] (*Included in Chapter 5*)

  In this paper, we propose ROS-Defender, a holistic approach to secure robotics systems, which integrates a Security Event Management System (SIEM), an intrusion prevention system (IPS), and a firewall for a robotic system. ROS-Defender combines anomaly detection systems at application (ROS) level and network level, with dynamic policy enforcement points using software-defined networking (SDN) to provide protection against a large class of attacks. Although SIEMs, IPS, and firewall have been previously used to secure computer networks, ROS-Defender is applying them for the specific use case of robotic systems, where security is in many cases an afterthought.

- Rivera, Sean et al. Auto-encoding Robot State against Sensor Spoofing Attacks. In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops(ISSREW)*. IEEE. 2019, pp. 252–257. [154] (*Included in Chapter 8*)

  In robotic systems, the physical world is highly coupled with cyberspace. New threats affect cyber-physical systems as they rely on several sensors to perform critical operations. The most sensitive targets are their location systems, where spoofing attacks can force robots to behave incorrectly. In this paper, we propose a novel anomaly detection approach for sensor spoofing attacks based on an auto-encoder architecture. After initial training, the detection algorithm works directly on the compressed data by

computing the reconstruction errors. We focus on spoofing attacks on Light Detection and Ranging (LiDAR) systems. We tested our anomaly detection approach against several types of spoofing attacks, comparing four different compression rates for the auto-encoder. Our approach has a 99% True Positive rate and a 10% False Negative rate for the 83% compression rate. However, a compression rate of 41% could handle almost all of the same attacks while using half the data.

- Rivera, Sean et al. Leveraging eBPF to preserve user privacy for DNS, DoT, and DoH queries. In: *Proceedings of the 15th International Conference on Availability, Reliability, and Security*. 2020, pp. 1–10. [171] (*Included in Chapter 7*)

  The Domain Name System (DNS), a fundamental protocol that controls how users interact with the internet, inadequately provides protection for user privacy. Recently, there have been advancements in the field of DNS privacy and security in the form of the DNS over TLS (DoT) and DNS over HTTPS (DoH) protocols. The advent of these protocols and recent advancements in large-scale data processing have drastically altered the threat model for DNS privacy. Users can no longer rely on traditional methods and must instead take active steps to ensure their privacy. In this paper, we demonstrate how the extended Berkeley Packet Filter (eBPF) can assist users in maintaining their privacy by leveraging eBPF to provide privacy across standard DNS, DoH, and DoT communications. Further, we develop a method that allows users to enforce application-specific DNS servers. Our method provides users with control over their DNS network traffic and privacy without requiring changes to their applications while adding low overhead.

- Rivera, Sean et al. Fast Monitoring for the Robotic Operating System. In: *Proceedings of the 25th International Conference on Engineering of Complex Computer Systems*. 2021 [177] (*Included in Chapter 6*)

  In this paper, we leverage the newly integrated extended Berkely Packet Filters (eBPF) and eXpress Data Path (XDP) to build *ROS-FM*, a high-performance inline network-monitoring framework for ROS. We extend the framework with a security policy en-

forcement tool and distributed data visualization tool for ROS1 and ROS2 systems. We compare the overhead of this framework against the generic ROS monitoring tools, and we test the policy enforcement against existing ROS penetration testing tools to evaluate their effectiveness. We find that the network monitoring framework and the associated visualization tools outperform the existing ROS monitoring tools for all robots with more than 10 running processes and that the monitoring tool uses only 4% of the overhead of the generic tools for robots with 80 processes. We further demonstrate the effectiveness of the security tool against common attacks in both ROS1 and ROS2.

- Rivera, Sean, Iannillo, Antonio Ken, and State, Radu. ”DiscoFuzzer: Discontinuity-based Vulnerability Detector for Robotic Systems”. 2020. [169] (*Included in Chapter 10*)

  Robotic systems continue their diffusion into society, accomplishing a myriad of physical tasks on our behalves. However, their safety-critical nature implies the vitality of testing their robustness. In this paper, we propose a novel fuzzing methodology that exploits the continuity of the physical world to automatically explore the input space and detect malfunctions in robotic software modules. The analysis of outputs includes the computation of the first and second derivative functions that can unveil anomalies in the behavior of the software. We implemented this methodology in *DiscoFuzzer*, the discontinuity-based fuzzer for ROS, and evaluate three different sampling approaches based on Monte Carlo, Chebyshev, and Spline methods.*DiscoFuzzer* detected 85 distinct vulnerabilities: 77 of the 89 previously known vulnerabilities and eight novel vulnerabilities previously undetected. The discontinuity analysis of *DiscoFuzzer* detected 41 more unique vulnerabilities than crash detection alone. Furthermore, we determined that the implemented sampling approaches were statistically equal in finding vulnerabilities, but each of them discovered vulnerabilities the others could not. The `chebfun` sampling is found to be the fastest in finding vulnerabilities. We determined that *DiscoFuzzer* provides a unique fuzzing solution for ROS systems to detect a wide

variety of security vulnerabilities with high precision.

- Rivera, Sean, Iannillo, Antonio Ken, and State, Radu. ROS-Immunity: Integrated Approach for the Security of ROS-enabled Robotic Systems. 2020. [170] (*Included in Chapter 4*)

  The Robotic Operating System (ROS) is the de-facto standard for the development of modular robotic systems. However, ROS is notorious for the absence of security mechanisms, only partially covered by recent advancements. Indeed, an attacker can easily break into ROS-enabled systems and hijacks arbitrary messages. We propose an integrated solution, *ROS-Immunity*, with small overhead that allows ROS users to harden their systems against attackers. The solution consists of three components: robustness assessment, automatic rule generation, and distributed defense with a firewall. *ROS-Immunity* is also able to detect on-going attacks that exploit new vulnerabilities in ROS systems. We evaluated our solution against four use-cases: a self-driving car, a swarm robotic system, a centralized assembly line, and a real-world decentralized one. *ROS-Immunity* was found to have minimal overhead, with only an additional 7-18% extra system power per robot required to operate it. Furthermore, *ROS-Immunity* was able to prevent a wide variety of ROS system attacks with a worst-case false positive rate of only 17% and a typical false positive rate of 8%. Finally, *ROS-Immunity* was found to be able to react to and stop attackers after at most 2.4 seconds, when confronted with unknown vulnerabilities.

1.2.2 As Co-Author:

- Lagraa, Sofiane et al. Real-time attack detection on robot cameras: A self-driving car application. In*: 2019 Third IEEE International Conference on Robotic Computing(IRC)*. IEEE. 2019, pp. 102–109. [152]

  The Robot Operating System (ROS) is being deployed for multiple life-critical activities such as self-driving cars, drones, and industries. However, the security has been persistently neglected, especially the image flows incoming from camera robots. In this paper, we perform a structured security assessment of robot cameras using ROS. We

point out a relevant number of security flaws that can be used to take over the flows incoming from the robot cameras. Furthermore, we propose an intrusion detection system to detect abnormal flows. Our defense approach is based on image comparisons and unsupervised anomaly detection method. We experiment with our approach on robot cameras embedded in a self-driving car.

Chapter 2

Background

"The greatest enemy of knowledge is not ignorance, it is the illusion of knowledge."

Daniel J. Boorstin

This chapter provides a background for readers unfamiliar with robotic systems or the security technologies employed in this thesis.

2.1 Robotics and Robotic Operating Systems

2.1.1 Robotics

Robotics is a diverse field involving the design, engineering, and use of robots to assist humans with a wide variety of tasks. Robotics is a sub-field of cyber-physical systems, which encompasses all devices with a physical component controlled by a digital component. In particular, robots are characterized by a particular degree of movement and autonomy to perform tasks [208], generally consisting of sensors and software that translate the digital world into the physical and the physical world into the digital. Physical robotic components may include mechanical 'arms,' such as those utilized in manufacturing or complex manipulative components used in drones and vehicles to facilitate movement controlled by software that governs how they interact with the physical world.

Translation between the physical and digital components in robots is facilitated by using a

plethora of sensors. Sensors are utilized to perceive the physical environment, using methods such as cameras, RFID scanners, LiDAR sensors, and microphones [19][161]. Sensors drive the difference between robots and machines, as, without a sensor, a piece of equipment cannot adapt to changing circumstances in its environment, an essential characteristic of a robot. Sensors do not have to be attached to a robot physically but can be merely present in their environment [50]. Utilizing sensor data, software controls the actions that the robot performs in the physical environment. It is unimportant if these behaviors are simple or complex, as complexity is a characteristic of the robotic system itself, not the definition of the system being a robot.

A popular example of the marriage between robots' cyber-physical components is self-driving cars, where large and complex physical components are combined with software to create autonomous tools that can perform tasks equally or more adeptly than humans. These systems utilize both physical cues from the environment observed via sensors and act upon those cues with adaptations in the car's functioning as determined by software.

Robots can be single autonomous systems, or they may be larger systems of interconnected, distributed components. A significant benefit of robotics is that multiple robots can be connected to perform more complicated tasks. As the complexity of robot-aided tasks rises, so does the complexity of the coordination between robotic systems. In many scenarios, there is a need for systems to require more than one robot is physically distributed in the environment, with these robots working together or utilizing information gained as a whole [143]. Of particular importance in this dissertation are the communication of robotics over a shared network to perform a common task, such as in a factory setting as discussed in Chapter 4 and Chapter 6.

The Importance of Robotics and Need for Robotic Security

Robots are observed plentifully in everyday life, with applications including, but not limited to, manufacturing, transportation, medicine, military, industrial, agricultural, and domestic products [95]. The penetration of robotics into society has vastly increased in the past 30 years due to the wide variety of tasks that they can perform[24]. In western society, humans interact with a plethora of robots and robotic systems every day, from interacting with domestic

robots within their homes, utilizing or interacting with vehicles running robotic systems, or benefiting from services run by robots by purchasing products manufactured or delivered by robots. Robotics held a market-value of $115 billion in 2019, with an expected CAGR[1] of 25% [196]. Between 2000-2016, 1.6 million jobs worldwide were lost due to automation by robots, with 20 million lost expected by 2030[138].

With the increasing penetration of robotic systems in everyday life, there is an increasing need for these systems to be as safe and secure as possible. Researchers have found vulnerabilities in commonly used robots, such as surgical robots, which could allow attackers to gain control of the actions of the robot during surgery and potentially harm the patient, and mechanical robots in factories that may allow attackers to gain control during ordinary functioning, potentially causing harm to their environments[52][41]. As the robotics market grows, as do the potential damages that these insecurities can cause, potentially impacting individuals, businesses, and the economy. However, there have been few effective approaches to address and prevent these vulnerabilities.

In this dissertation, the concerns with robotic security measures addressing robotic operating systems are discussed and addressed. This work aims to provide comprehensive tools to secure robots and robotic operating systems toward avoiding damages in the physical and digital world.

2.1.2 Robotic Operating Systems

The Robotic Operating System (ROS) is a collection of software libraries that enables communication of both (abstracted) hardware and (pure) software components to develop robotic systems. It is predicted that ROS centered systems will make up the majority of robotic systems within the next five years, both in commercial and academic settings [149]. Even among non-ROS systems, many design similarities exist wherein similar methods can be applied [22]. ROS is an open-source project, and it has a vibrant community with thousands of developers and over nine thousand unique packages available[2].

Usually, these packages extend ROS core functionalities by implementing commonly

[1] Compound Annual Growth Rate
[2] https://metrics.ros.org/

used robotic software modules such as hardware drivers, robot models, datatypes, planning, perception, localization, simulation tools, and other algorithms. Of particular mention is the ROS-Industrial [172] consortium: it consists of 78 international industries that chose to extend and adopt ROS for their industrial robots.

The ROS project already includes guidelines for testing [173] by applying existing libraries, *i.e. gtest* for C nodes and *pytest* for Python nodes, for simple unit testing. A tester should design and implement every test in the appropriate language with test inputs and oracles for every behavior she want to check.

ROS Architecture

ROS provides a framework to applications consisting of independent computing processes called **nodes**, with the help of a **master node** acting as a global **namespace**, a parameter server acting as a repository of globally shared data, and a **middleware layer** providing a consistent set of interfaces for software development and hardware. We distinguish three layers in ROS architecture: Application, Middleware, and Operating System. The application layer promotes faster development, fault isolation, modularity, and code re-usability. All processes are abstracted as nodes within the middleware layer, with communication between processes being completed by either topics(unidirectional) or services(bidirectional). They facilitate the communication between nodes based on two abstractions: **topics** and **services**. All processes run on top of a UNIX operating system. In ROS, nodes communicate through **messages**, which define clean and consistent interfaces.

The ROS system center is the **roscore** utility that contains the ROS master, the ROS parameter server, and the node **rosout**, the standout logging node. The master node is the central communication hub that tracks all the offered topics and services and maintains a dictionary mapping of all nodes' location, which nodes provide which topics/services, and on which ports those topics and nodes are located. Unfortunately, the master node is unable to enforce this map creating many vulnerabilities [164] [56] for ROS system. The parameter server acts as a global variable repository for the nodes. The parameter server is a shared repository between nodes. The rosout node is an implementation of stdout for the rest of the system. Any node can become a publisher to rosout to output messages. Without these

components, the ROS system cannot communicate or create new nodes; thus, it should be assumed that they always exist in a ROS system. Both the master and the parameter server are implemented using XMLRPC, a stateless HTTP-based protocol, which allows data to be stored by key in a central location.

ROS uses nodes as a high-level abstraction for defining individual processes running within its framework. Each node is designed to be a self-contained process to control part of the robot's operation and must implement several shared components to integrate with the rest of the system. These components include the ROS slave API, the internal ROS protocol, and the ROS command-line interface. The ROS slave API handles all interactions with the ROS master as well as with other ROS nodes. Each node in a ROS system must have a unique name[192]. If two nodes have the same name, the older node is shut down by the master node.

Nodes communicate with each other using messages. Messages are simply a data structure that contains the typed field, which can hold a set of data, and that can be sent to another node. Besides the standard primitive types such as integer, floating-point, or Boolean, the developer can also build its message types using these standard types.

A ROS topic is an implementation of a channel in a publish-subscribe model. Topics are the primary form of communication for the ROS system. Each topic is defined as a unidirectional, many-to-many publish-subscribe model. Topics act as a named bus that any node can join as either a publisher or a subscriber, and they define the shared communication path with the ROS message interface. In order to create a topic, a node informs the ROS master by providing the name of the topic and what data type the topic will use. The master informs the node about all of the other publishers and subscribers to the node. The ROS master maintains an internal list of topics and which nodes are currently publishing and subscribing to them. When a ROS node becomes a publisher on a topic, it opens a port for subscribers to connect to that specific topic. When multiple nodes are publishers on a topic, the subscriber must communicate with all of the publishers. There is no access control for topics beyond the data type MD5 hash. An example is shown in Figure 2.3, where the ”Camera” node sends messages to the ”Images” topic. The messages in the topic are received by the

”Storage” node and the ”Processing” node. The ”Storage” node depends on the underlying Linux file system to provide access to the storage location. The publisher-subscription model is designed to be modular at a fine-grained scale and is suitable for distributed systems.

Services handle bidirectional node communication through the use of XMLRPC function calls. Once a node decides to provide a service as part of its interface, it opens a unique port for that service to which any other node may communicate and inform the master of the service's location. While services can be used as one-off function calls, they also provide mechanisms for continuous communication through persistent connections.

The ROS middleware layer composes the common communication threads between nodes. ROS also includes a communication system, TCPROS/UDPROS, that requires a master process and presents extensions of the TCP and UDP protocols, respectively. TCPROS is preferred, as UDPROS is still in development. Developers using UDPROS must implement what they need on a case by case basis. All TCPROS headers provide both an MD5 hash of the message type and its name, using the hash to confirm that the node is sending the same type of message. Each node must have an internal implementation of either TPCROS [194] or UDPROS [195]. As an alternative to TCPROS and UDPROS, ROS supports **nodelets**, which realize non-serialized data transport between nodes in the same process by passing a pointer. The communication between nodes, topics, and services is represented by a dynamic graph called *ROS graph*.

Figure 2.1: The Universal Robot's node and topics

As an example, Figure 2.1 shows a node and its topics from the Universal Robot package [3]. The `arm_controller_spawner` plans the movement of a joint to reach a determined position and makes the joint perform this movement. The node subscribes to one topic in which a message consists of different components (inputs), *e.g.* angular position, and effort. It publishes a single topic in which a message consists of different components (outputs):

[3]http://wiki.ros.org/action/show/universal_robots

the current angular position, the velocity, and the acceleration. We based the example in Figure 2.1 on a simplification of the Universal Robot package.

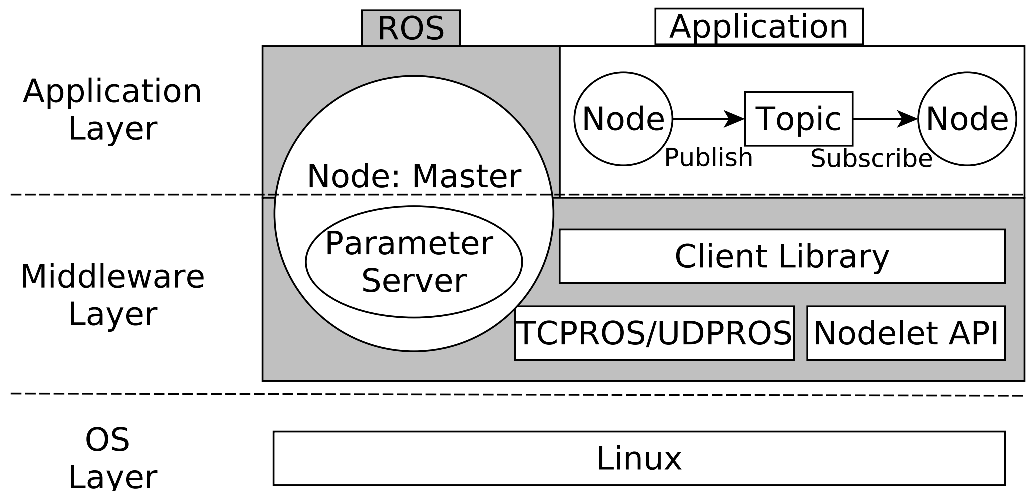


Figure 2.2: ROS Architecture

ROS2

The second version of ROS, namely *ROS2*, was released in 2018. The goals of ROS2 are to enhance the performance of multi-robot communications and provide real-time capabilities. The novelty in ROS2 is the modification of the middleware layer, which uses the Data Distribution Service (DDS) as its networking middleware. DDS supports the distributed nature of the robotic data and Quality of Service (QoS) based configurations. Compared to ROS1, it simplifies complex network programming by implementing a publish-subscribe system for sending and receiving data, events, and commands among the ROS nodes.

ROS2 offers the following advantages regarding ROS1:

- Real-time requirements: initially designed for single robot control with no real-time and using reliable connections.
- Distribution: ROS1 uses a centralized discovery, the ROS master (single point of failure). In contrast, ROS2 is fully distributed, including discovery.

- QoS: uses quality-of-service settings to handle lossy networks and efficient intra-process communication.
- New use cases operating in distributed environments: autonomous vehicles, multi-robot swarms,...etc.

However, as ROS2 is still under development, it is still unclear when a complete functional switch from ROS1 to ROS2 will occur. Presently, a helper node called *ros1-bridge* allows developers to use the hybrid systems of ROS1 and ROS2 to ease the transition.

A Typical ROS system

An typical ROS system is made up of 'multiple' packages (based on robot purpose) per robot [141], with each package containing an average of 3.8 nodes [156], and each node communicating on 5-8 topics [87]. These figures are measured on ROS1 but can be assumed to apply to ROS2 as well.

For example, as shown in Figure 2.3, the "Camera" node sends messages to the "Images" topic. The messages in the topic are received by the "Processing" node and "Storage" node. After the messages are received, the "Processing" node performs a standard image processing task of simple object recognition while the "Storage" node keeps a log of the data. The publisher-subscription model is designed to be modular at a fine-grained scale and is suitable for distributed systems.

Figure 2.2 briefly illustrates the system models of the Robotic Operating System(ROS). ROS is designed to provide a consistent set of interfaces for software development and hardware, similar to the interfaces provided by an operating system.

Centralized and Decentralized ROS systems

This dissertation assumes that two main design paradigms are applied to implement a multi-robot system with ROS. The primary difference is the number of master nodes in the system, with centralized systems utilizing only one master node while decentralized systems more than one. Both design paradigms allow for unique functionality while carrying their own challenges.

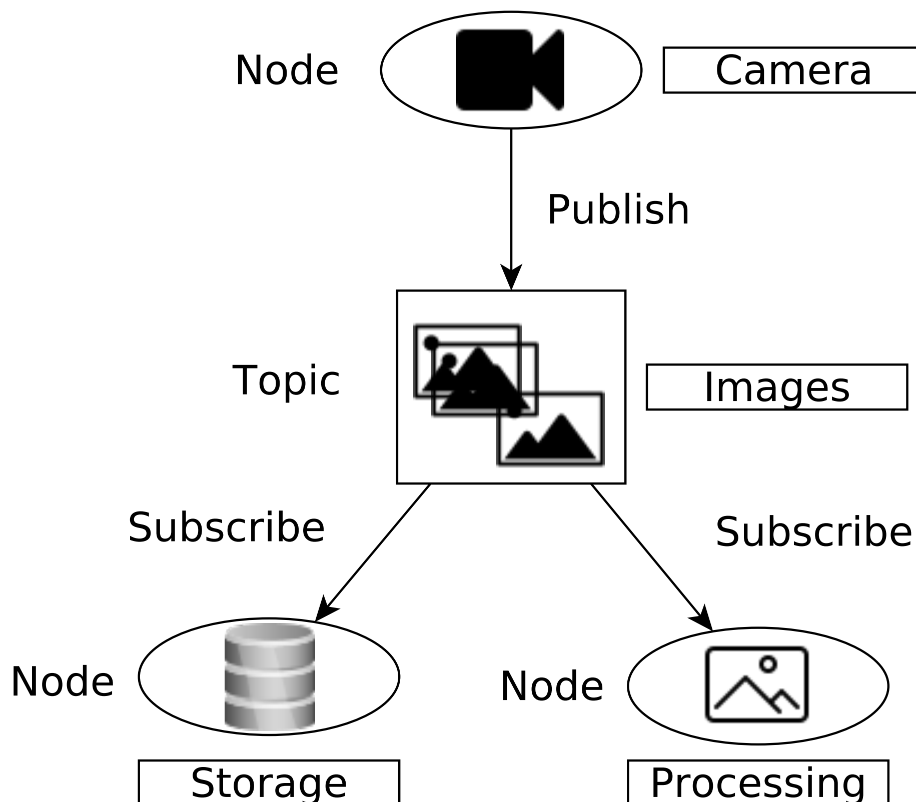


Figure 2.3: Example of a simple ROS system, with camera data being split between two components; object recognition and logging .

Centralized ROS systems include all systems where there is only one ROS master node for all robots in the system. For users interacting with the system, the master node is the singular point of contact. All logs and priority alerts will go to the master for communication to the user. Robots interacting with the system are assumed to communicate with the centralized ROS master node.

The following assumptions hold about robots in a centralized system:

- It is assumed that all other robots are aware of the master node and that it is the only ROS master to which they can communicate.
- It is also assumed that robots must communicate with the ROS master for functionality.

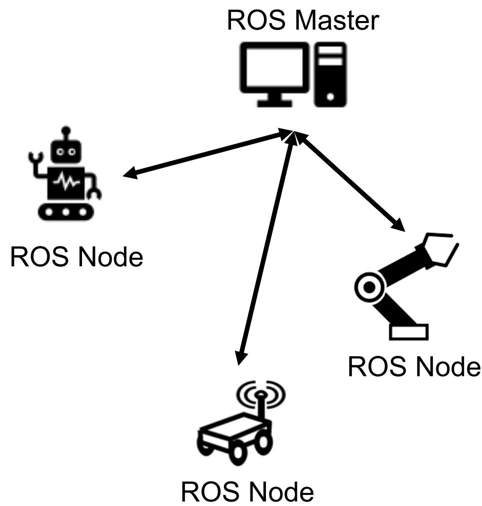


Figure 2.4: ROS Centralized System Design with a single master node and communication channel from the master node to the other nodes.

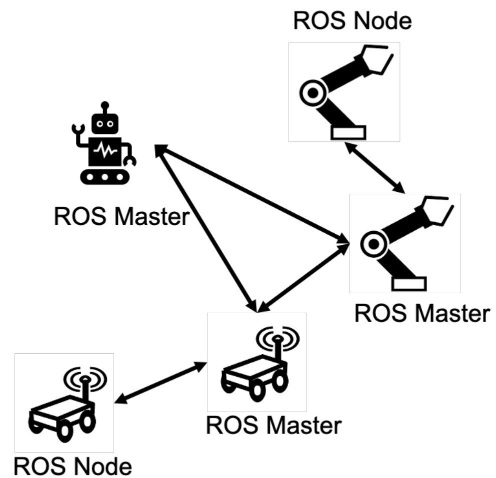


Figure 2.5: ROS Decentralized System Design with multiple master nodes and hierarchical communication model among the robots.

- While robots may have the ability to connect over channels not monitored by the master, this communication does not replace communication with the master: all robots must have a direct, non-proxy connection with the master node.

These assumptions imply that every single robot in a connected system can reliably communicate to a centralized system (or robot). The ROS master can act as a single trusted repository word encrypted communication identity validation and the distribution of new firewall rules. While robots may not always communicate with the master, especially once the system has reached a steady-state, the master can always initiate a connection.

The centralized architecture is the most common type of ROS infrastructure as it is the easiest to set up and the only infrastructure natively supported for ROS1. However, this infrastructure carries with it several core challenges preventing it from becoming a universal solution. The first such challenge is poor network connectivity that breaks the master's control over other robots, which can place the system in an ambiguous or dangerous state. In such a state, part of the system is controlled by the master, and part of it is silently oper-

ating as if the master still has control, even though that control has been lost. Secondly, it suffers from verification issues. A common underlying ROS1 security vulnerability, wherein the master lacks any way to verify the state of the ROS system[165], is magnified in cases where nodes from two (or more) robots are communicating over a separate network connection that the master has no way of accessing. As compromised robots will communicate the same statuses as their un-compromised counterparts, and there is no ability to cross-validate using a higher ring of security like root or kernel, this significantly increases the difficulty of detecting compromised robots. Finally, while ROS is highly interoperable, there are still challenges when integrating hardware components from different manufacturers. These components require either many difficult-to-maintain standard libraries or require the user to set up parallel systems. Depending on a user's specific needs, they may be forced to forgo a centralized approach for a more decentralized or partially decentralized one.

A decentralized system is one where there are multiple master nodes with a lack of hierarchy. This lack of hierarchy could be due to an impossible-to-establish or impossible-to-verify hierarchy. This includes any system where there are two co-equal masters for different components or any system in which parts of a robot cannot communicate with the absolute master and rely on an intermediary. While ROS2 has direct support for such systems, ROS1 instead relies on packages and other workarounds. Thus, a decentralized ROS1 system cannot easily apply any cryptographic key-based system. Decentralized systems are most common in swarm robotics and in systems with unreliable networking, although other examples may exist.

When dealing with decentralized ROS systems, the core assumption is that no robot holds absolute authority within the system. Decentralized systems may include systems designed not to have one central robot or systems designed to have one central robot where the robot cannot reliably communicate its authority to all others.

Chapter 3

State of the Art

"An expert is a person who has made all the mistakes that can be made in a very narrow field."

Niels Bohr

In this chapter, the current state of the art in robotic security will be discussed. Analysis of the security approaches for ROS are the focus as the proposed solutions of this dissertation focus on ROS security; however, general robotic security approaches are discussed as well.

3.1 General Robotic Security

The field of robotics security has been broadly explored for decades. It focuses primarily on safe, consistent failure (i.e., ensuring robots always enter a "fail-safe" state) and tends to look at components that all robotic systems share, such as sensors, control algorithms, and networking.

Robotic systems security broadly encompasses several types of vulnerabilities [82] [118][111][104][92][1
Each robotic component has potential for vulnerabilities, and even more vulnerabilities can be created through the interactions between various components. Even the data fed into robots is a vector for potential compromise and damage. The field of general robotic system security and their solutions is too broad to analyze in any dissertation; therefore, this dissertation focuses on the robotic security of commonly used robots that utilize ROS systems and their components.

3.1.1 Common Robotic Systems

In this dissertation, the focus is maintained on four types of robotic systems (and their components) that are broadly representative of the field in ROS security research. Each type has very disparate design paradigms and security challenges, which gives security researchers a clear target for which to design solutions. These four types include autonomous vehicles (self-driving cars), robots utilized in factories, homogeneous swarms, and drones.

Autonomous vehicles are perhaps one of the most trendy topics of the last decade, with notable security vulnerabilities being exposed regularly. Koscher *et al.*[17] presented composite attacks in automobile systems by infiltrating in the electronic control unit and highlighted the lack of detection or enforced protection mechanisms of essential services such as diagnostic and reflashing. Furthermore, the same authors stressed the importance of securing the vehicle bus where third-parties components are automatically trusted, leaving space to attackers.

Checkoway *et al.*[18] also targeted modern automotive systems and demonstrated the feasibility of remote exploitation through several attack vectors (*e.g.*, CD players, and Bluetooth). One of the final recommendations consists of hardening the underlying OS because it cannot be assumed that the attack surface will not be breached.

Both Eiza *et al.* [73] and He *et al.* [76] perform broad surveys of the security challenges related to cars. The authors focus primarily on the potential damages that can be inflicted on the vehicle operator in the event of a compromise and some potential avenues. The survey concluded that the greatest risk of compromise comes from sharing a network between life-critical components and consumer-facing applications.

Building on all of this vulnerability research, Chowdhury *et al.*[163] wrote an extensive survey of all known vulnerabilities and the current best countermeasures available. They focused their efforts on the vehicle's network and ensuring that critical components cannot be attacked merely through the compromise of other less critical components. The paper also outlines all of the components of self-driving cars and highlights other research that effectively secures them.

Robots utilized in factories are perhaps the most prominent example of ROS systems,

as well as the primary focus of the largest ROS consortium, ROS-Industrial [172].

Quarta *et al.*[84] analyzed the reference architecture and real-world industrial robotic systems to assess their security capabilities and formalized security challenges to address in the short-, medium-, and long-term. Among them, attack detection and system hardening are listed as countermeasures to security breaches. Furthermore, the authors highlighted that the typical assumption of industrial systems that internal networks can be trusted is not realistic. Filtering inputs and messages in the robotic system network should be considered from the first phase of design.

Pogliani *et al.* [151] discusses the nature of robotic security in 'collaborative' industrial settings where industrial robots work near humans. The authors discuss the current drawbacks of industrial robots' security requirements and how the underlying assumptions relating to the separation of humans and robots no longer apply. They then propose several new solutions to help address this vulnerability and keep humans safe in a more collaborative environment

Homogenous swarms are considered as distributed groups of identical robots that work together to perform a common task [1]. Swarm robotics is challenging to secure due to its unique vulnerability structure compared to conventional systems. It is regarded as a field of interest in the coming years due to these unique characteristics [27].

Ferrer [101] addressed swarm robotic systems and their need for security in cooperation activities. He advocates the uses of blockchain technology to provide security solutions to the swarm robotics research field. Ferrer *et al.*[142] proposed a model based on Merkle trees, where the robots in the swarm exchange cryptographic proofs to demonstrate their integrity. In such a way, even if robots do not know the overall mission goal, they can cooperate and perform operations correctly.

Strobel *et al.*[127] also exploited blockchain technology and decentralized programs (smart contracts) to secure coordination in swarm systems and to detect Byzantine robot.

Similarly, Cameron *et al.*[96] exploited blockchains to develop a multi-chain system to detect Byzantine actors in swarm robotics systems.

[1] Heterogeneous swarms are considered outside of the scope for this dissertation.

Drones have become incredibly popular in the consumer market, followed shortly by in-depth analyses of the cybersecurity concerns on these vehicles [81].

3.1.2 Robotic Components and Vulnerabilities

Attacks on robotic components are of utmost importance, especially those that influence the robot's planning and functionality. Of particular importance in this dissertation are sensors and control algorithms.

Sensors include cameras, LiDAR , microphones, and all other components that inform the robot of its state in the physical world. Software and hardware solutions have been proposed to countermeasure attacks on camera and LiDAR of automated vehicles using commodity hardware [48]. Spoofing attacks have been performed by Shin *et al.* [88], who proposed a spoofing attack causing the illusions to appear closer than the location of a spoofing device. Similarly, Davidson *et al.*[55], proposed a spoofing attack against LiDAR and introduced a method for defending against such an attack on optical flow sensors, using the RANSAC [2] algorithm to synthesize sensor outputs. Choi *et al.*[99] proposed an attack detection framework that identifies external, physical attacks (including sensor attack, actuation signal attack, and parameter attack) against robotic vehicles on the fly by deriving and monitoring Control Invariants (CI). Kapoor *et al.*[109] detect spoofing attacks against an automotive radar system by effectively verifying physical signals in the analog domain.

Guo *et al.*[75] proposed robot intrusion detection systems (RIDS) that can partially detect sensor attacks. It can detect, pinpoint, and quantify sensor attacks when not all sensors are simultaneously corrupted. RIDS consists of four modules: a monitor, a multi-mode estimation engine, a mode selector, and a decision-maker. The monitor collects and sends data to the estimation engine. The estimation engine generates a set of estimation results under different hypotheses and their corresponding likelihoods. The mode selector accepts the more likely hypothesis. The decision-maker leverages the estimation results from the accepted hypothesis to detect attacks [75]. The same authors proposed the same anomaly detection approach to detect misbehavior in mobile robots' sensors that actively influence robot behavior and cause damages in the physical world[103].

3.2 ROS Security

The security exploration of ROS systems has been historically under-looked, with large scale research only beginning in 2013 [29]. However, it was not until 2017 that researchers began exposing ROS security vulnerabilities, presented in *Security for the Robot Operating System* [72]. The authors highlighted the security issues with several possible attack vectors on a ROS application, such as unauthorized Publishing (Injections), unauthorized data access, and denial of service (DoS) attacks on specific ROS nodes. They showed how to secure ROS on an application level and describe a solution integrated directly into the ROS core. They extended this research into RosPenTo, a semi-automated tool for testing ROS[186] that injects fake messages into the system to demonstrate the consequences of the lack of security in ROS. They show that an attacker (1) can easily query the master for sensitive information, and (2) can easily impersonate a subscriber node. . Following ROSPenTo, this research was further extended to both red-teaming [168] and penetration testing [164] the robotic operating system, formalizing the approach for security of ROS robots.

Robotic platforms suffer from security threats in the communication link and the applications that are unique to them [93]. Several relevant security flaws can be used to take over and command the robot. An attacker could stop the components that control the robot, such as the camera, sensors, or legs, to immobilize the robot [116].

The formalization of ROS security for robots is extended by several case studies that evaluate the theoretical framework on existing robotic systems. These studies aimed to find vulnerabilities beyond the theoretical, particularly those that may affect humans during abnormal functioning. One such case study is the Pepper robot, where a detailed analysis of a ROS system's security is provided in *Adding Salt to Pepper* . The authors performed a formal security analysis of the robot but did not consider the security of the ROS components. They found many traditional vulnerabilities (weak passwords, missing authentication), but they did not consider any potential ROS attacks. While this shows that ROS systems still have typical security concerns, an in-depth analysis was not provided on ROS-specific components' security.

The second case study was of a teleoperated surgical robot running ROS [41].

To address the security problems plagued by ROS systems, several solutions have been proposed. The author of Secure Communication for the Robot Operating System[68] proposed a secure communication between ROS nodes via TCP and UDP using Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). This means that the possession of a trusted certificate is necessary for publishers or subscribers. Similarly, in SROS: Securing ROS over the wire, in the graph, and through the kernel [64], the authors proposed SROS [2], a library for ROS ecosystem to support modern cryptography and security measures to address existing vulnerabilities. In SROS, all network communication is encrypted using Secure Sockets Layer (SSL), specifically TLS. The encryption is done through the use of Public Key Infrastructure (PKI), where each ROS node is provided an x.509 certificate, equivalently an asymmetric key pair, signed by a trusted certificate authority. These results indicate security vulnerabilities in ROS, requiring additional libraries to ensure security in vital ROS systems. Rodríguez-Lera *et al*. [122] proposed to solve ROS nodes communication problems by encrypting TCPROS messages at the application level. They analyzed the behavior of three well-known encryption algorithms such as 3DES, AES, and Blowfish, and their effects on the robot's autonomy. They showed how to secure ROS on an application level and describe a solution integrated directly into the ROS core. Thus, the implementation of ROS itself must be changed.

However, no singular solution is perfect, allowing for the extension of frameworks to monitor ROS systems and detect vulnerabilities. One of the first of these frameworks, ARNI, was proposed by Bihlmaier *et al*. [53, 32], a framework to monitor and introspect large ROS systems at run-time to find configuration errors and bottlenecks. Its primary purpose is to collect and visualize information about the message flow inside the distributed network system. Additionally, it provides information about system resources such as CPU and memory for hosts and individual ROS nodes. ARNI allows visualizing using a dashboard. Additionally, it proposes countermeasures that can be taken to ensure the continued functionality of the ROS network, such as detecting a violation of a known state.

[2] `http://wiki.ros.org/SROS`

Another solution included Drums, a lightweight distributed monitoring system resources, and a debugging tool for robot systems proposed by Monajjemi *et al.* [38]. System resources are PIDs or sockets of ROS nodes. Drums are used as a component for testing, debugging, and run-time quality-of-service monitoring. However, these network monitoring tools do not provide deep network information such as net-flow data in a network system or new communication means.

Beyond the traditional security systems, solutions for securing ROS nodes have been proposed, particularly a property-based testing framework for ROS implemented by Santos *et al.*[125]. This first approach aims to automatically generate test scripts for property-based testing of various configurations of a ROS system. Their approach looks for sequences of messages that either crashes the target node or violate a previously specified property. An automatic test generation method builds the property-based tests from configuration models extracted by a static analysis framework. Further, model-based testing is currently the most used technique that formalizes the system's behavior and then checks its correctness. Zander *et al.*[30] presented a methodology that allows the execution of the model before the actual implementation occurs. Additionally, Saietti *et al.*[49] exploits Petri nets to model the behavior of cooperative robots and propose a test case generation algorithm to cover the whole model.

All of the previous systems were developed primarily for ROS1. The security for ROS2 is a less established field; however, Kim *et al.*[112] provided an exploratory look into the challenges and performance considerations for ROS2 systems with the secure DDS standard. They find that the current implementations of secure DDS add an overhead of around 100%, which is still usable for critical computing; however, they did caution that a more efficient implementation of the standard was possible.

3.2.1 Security of the ROS network stack

The master plays a critical role in ROS; without it, nodes would not be able to find each other, exchange messages, or invoke services. If the master node is compromised, it is functionally identical to the whole system being compromised. Communication with the master is done

over a stateless HTTP-based protocol, lacking security support in authentication, integrity, confidentiality, and availability. The lack of these security services allows a node to (1) publish data for an arbitrary topic or (2) subscribe to any topic. The former may be misused to inject data into an application to disturb its operation, such as causing unpredictable damages [72]; while the latter allows a node to receive any data published for that topic.

The parameter server is also implemented using XMLRPC, meaning that it suffers from the same lack of protection as the master. By default, all ROS nodes can change any global parameters without authentication or verification. The parameter server contains necessary configuration settings for the robot, and alterations to the parameters can drastically alter the system's behavior. For example, in [197] it was shown how to change the PID [4] control gains stored on the parameter server, causing the motors to respond to environmental changes. Changes such as these would lead to unexpected behavior for an operator or any algorithmic assembly process.

Nodes have open ports accepting connections associated with different topics or services. This communication is established mainly with the help of TCPROS, relying on TCP. Again, with the exception of an MD5 hash, there is no protection on these packets, allowing an attacker to create numerous problems, such as learning confidential information and injecting, modifying, and replaying packets. Nodelets are wrapped in a nodelet-manager node, which allows them to communicate with shared pointers to other nodelets. Any communication between nodelets and nodes is still done through the TCPROS connection.

SROS

A recent effort into addressing ROS's security concerns is SROS, an experimental security suite designed to harden ROS systems against several standard classes of attacks [200]. It is structured around three security concepts; Transport Security, Access Control, and Process Profiles [64]. Transport Security replaces ROS communications with TLS using `X.509` certificates, which provides confidentiality for communication between nodes and protection from man-in-the-middle attacks. Access Control ensures that ROS nodes cannot make unauthorized changes to the function of the ROS graph. At the moment, however, the Access Control level does not have a concrete implementation. There are two competing

standards for this level: extending the `X.509` certificates with PKI metadata or using an online arbiter[200]. Process Profiles hardens the ROS nodes themselves through the use of process isolation and sandboxing. It does so by providing an AppArmor profile fit for ROS nodes. [180] AppArmor is a Mandatory Access Control (MAC) implementation for Linux. Its goal is to define process access control for Linux systems. AppArmor functions by defining which paths a process is allowed to access and what actions the process is allowed to perform. AppArmor confinement is provided via profiles loaded into the kernel, typically on boot.

While SROS addressed some of the security concerns with ROS, it still has several limitations. Use of TLS is as effective as the public key certificates management and protection; it also does not protect against compromised nodes. The policy definition component of SROS has still not been implemented and depends on either defining a set of static rules or leaving the system on at all times. This means that a system cannot react to compromised nodes and that at any level of system complexity, it is impossible for the developer to maintain the static rules reliably. If an insider threat were to sign a malicious node with an invalid certificate, they would be able to completely negate the policy protections from the `X.509` certificate. Additionally, the SROS system would not be able to adequately react to the malicious node, potentially opening up the system to the same devastating attacks that would affect a typical ROS system. Finally, the use of AppArmor confines the type of networking policy that can be enforced. Specifically, the networking control provided is too coarse-grained. For example, one can not restrict binding on specific ports or integrate with a firewall.

ROS2 is intended for far more applied situations than ROS1 and attempts to incorporate security through the DDS secure standard and code verification project[112] by taking advantage of the OMG DDS security standards[126]. These two solutions fill a similar role for SROS in ROS1; however, they are designed to be easier for users to take advantage of. This is because the DDS implementation is designed to be invisible to ROS2, so the difference between SROS2 and ROS2 from the user's perspective is only a simple configuration change. It is crucial to keep in mind, however, that SROS2 is not the final word on ROS2

security and still contains many security flaws that require redress[117][100].

3.2.2 Vulnerabilities in ROS

Vulnerability	Description
Node Attacks	This category of vulnerabilities contains all of the attacks against the ROS node infrastructure itself. The most common attack of this type is launching a copy of a critical system node that replaces the normal function node, granting the attacker access to that node's data.
Message Alteration	This category of vulnerabilities contains all of the attacks against the ROS messages, such as changing the messages' contents or by resending copies of already sent messages.
Topic/Service Attacks	This category of vulnerabilities contains all of the attacks against the ROS topic and service infrastructure, including altering the types of messages used or interfering with the provider information.
Communication Attacks	This category of vulnerabilities contains all of the attacks against the TCPROS and UDPROS frameworks. It covers techniques like packet reassembly attacks, jumbograms, and header formatting attacks.
Configuration Attacks	This category of vulnerabilities contains all of the attacks against the ROS Master and parameter server and any attacks against other configuration routines such as launch files.
Node Software Bugs	This category of vulnerabilities is a catchall for all vulnerabilities within the ROS node code that are not part of the ROS middleware itself. This is the largest class of vulnerabilities.

Table 3.1: All currently relevant types of vulnerabilities that are specific to the ROS middleware and its components.

Vulnerability	Description
Arbitrary Code Execution	This category of vulnerabilities contains all vulnerabilities where an attacker is able to run software of the attackers choosing on the system. These vulnerabilities grant the attackers broad versatility in what they can do and is a very damaging vulnerability.
Operating System Attacks	This category of vulnerabilities contains all vulnerabilities where an attacker attacks the underlying operating system. This is the most critical class of vulnerabilities as compromises to the OS can cause massive damage.
Third-Party Binary Attacks	This category of vulnerabilities contains all vulnerabilities relating to any other software running on the robot that's not related to ROS. This is the most common source of vulnerabilities for the robot, as there are usually more third party programs then ROS nodes or other programs.
Physical Attacks	This category of vulnerabilities contains all vulnerabilities where an attacker attacks the robot hardware itself. This can mean physically adding or removing hardware, blocking or damaging the robots, or even more esoteric circuit attacks.
External Interception	This category of vulnerabilities contains all vulnerabilities where an attacker is able to remotely listen in on and change communication coming from beyond the robotic systems. Such attacks are more common on robots with enabled wireless communication for obvious regions; however, they do occur on hardwired ones as well.
Data attacks	This category of vulnerabilities contains all vulnerabilities where an attacker is able to compromise the data sources a robot depends on to perform its tasks. This does not include direct attacks on the sensors but covers all other attacks.
Sensor Attacks	This category of vulnerabilities contains all vulnerabilities where an attacker is able to compromise the sensors the robot depends on to perceive the outside world. This can mean physical interference with the sensors, or interference with what the sensor can perceive, or even the generation of false signals.
Remote Component Control	This category of vulnerabilities contains all vulnerabilities where an attacker is able to manipulate any of the non-ROS components that the robot uses. This includes things like connected manipulators, debug interfaces, or even things like conveyor belts.
Power Attacks	This category of vulnerabilities contains all vulnerabilities where an attacker interferes with the power going to the robot. This can introduce a denial of services or even introducing undefined behavior during 'brownout' conditions.
Cloud Attacks	This category of vulnerabilities contains all vulnerabilities where an attacker interferes with any cloud providers that the ROS system is using. This tends to be out of the robot owner's control; however, it can still introduce issues.
Certificate Attacks	This category of vulnerabilities contains all vulnerabilities related to X.509 certificates or their authority. Any attacks here require all the robot's cryptographic components to be replaced, which requires a full system overhaul.

Table 3.2: All the types of vulnerabilities not caused by the ROS middleware, but can nevertheless still effect ROS robots.

Part I

Integrated Security

Chapter 4

Integrated ROS Security

”As we’ve come to realize, the idea that security starts and ends with the purchase of a prepackaged firewall is simply misguided.”

Art Wittmann

4.1 Introduction

This chapter presents the completed integrated system for securing and monitoring ROS in its totality. Dubbed *ROS-Immunity*, this tool acts as an immune system for ROS, defending against known threats, identifying new threats, and mitigating compromised systems’ damage. The chapter describes the culmination of all work for this dissertation, an integrated system with a small overhead that allows ROS users to harden their systems against attackers, especially those who have access to internal interfaces between ROS nodes. The security challenges of both centralized and decentralized ROS systems are analyzed.

In the age of technology pervasiveness into everyday life, security is of vital importance. However, designing a ’good’ security system is historically very difficult. Every system has different requirements, and the addition of security systems generally makes them more complicated, while complexity is simultaneously the enemy of security [6]. Whitten & Tygar, 1999, propose that the key problems for a security system are user motivation, abstraction,

lack of feedback, the *barn door* problem[1], and the *weakest link* problem [2][6]. In this dissertation, these key problems are used to guide the development of a security system for ROS. Here, a good security system is proposed as one where:

- Users can easily secure the system to their specifications.
- Users are aware of the current security status of the system.
- Limited engagement with the security system, i.e., the security system should have low upkeep.
- Users are informed of the security issues that need addressing, without being overwhelmed by false-positives.
- The structure of the security system closely matches the structure of the real system.
- The system can be placed back into a secure state reliably after a breach.
- The system does not burden regular operation.

This dissertation focuses on applying these guidelines to design a security framework for ROS systems. Due to the prevalence of ROS systems and significant security issues, creating such a system is of vital importance. Previous work in the area of ROS security provided several critical insights that culminated in this tool. These insights were:

- making the configurations speak the same language as ROS,
- building the system with the ability to communicate issues that it could not address to users efficiently, and
- ensuring that the system had a highly modular design.

Making the configurations speak the same language as ROS addresses the concept of abstraction, sparing users from having to maintain two separate languages. It also removes

[1]The *barn door* problem is the concept that once a secret has been exposed or a system has been left unprotected, there is no way to be sure that it is not already affected by an attacker. [6]

[2]The *weakest link* problem states that "the security of a computer is only as strong as its weakest component". [6]

the risk of translation errors, which causes a mismatch between what the security system is *supposed* to address, and what it *truly* addresses. Building a system with the ability to efficiently communicate issues that it could not address on its own is required due to the safety-critical nature of robotic systems. If a security system is built without the ability to inform a user of potential compromise immediately, significant harm can occur. Further, the system must be able to place the robot into a fail-safe state to protect users until it can confirm security again. Lastly, a modular design is vital, particularly for ROS systems, due to ROS's highly-modular nature. Without such modularity, users would be forced to exert far more effort to integrate the security system with their personalized ROS configuration.

These guidelines and insights were used to design *ROS-Immunity*, a comprehensive security system for ROS. *ROS-Immunity* provides a lightweight security solution for ROS capable of covering the whole chain from discovering vulnerabilities to protecting from attacks. *ROS-Immunity* consists of three components: robustness assessment, automatic rule generation, and distributed defense with a firewall. The robustness assessment discovers new vulnerabilities of the target through the combination of different testing and analysis techniques. Then, these vulnerabilities are encoded in a domain-specific language and automatically fed into the distributed firewall. Finally, the firewall runs on the target system's robots and can detect and block malicious messages or other malicious behavior.

In this chapter, four use-cases are discussed for ROS systems where multiple ROS nodes are distributed over the network to showcase the solution's abilities. First, a self-driving car, where components are spread throughout the vehicle, is targeted. Then, it targeted MONA[132], a swarm robotic system where distinct robots collaborate to reach a common goal. Lastly, centralized and decentralized robotic clusters are addressed, with both a centralized and a real-world decentralized assembly line based on ARGoS[21], where many task optimized robots work in a sequential network.

Throughout this chapter, the underlying technologies of *ROS-Immunity* will be discussed. Each system within *ROS-Immunity* results from the advancement of tools and techniques discussed later in this dissertation. Robustness assessment is conducted utilizing many tools, including *ROSploit* (Chapter 9) and *DiscoFuzzer* (Chapter 10). The framework

for the automatic rule generation component of *ROS-Immunity* was based on the design in ROS-Defender (Chapter 5) and *ROS-FM* (Chapter 6). The firewall component is a key contribution built with a combination of *ROS-FM* (Chapter 6) and a state-based autoencoder (Chapter 8). These tools are combined, adapted, and extended to build *ROS-Immunity*.

4.2 ROS System Designs: Centralized and Decentralized Systems

There are two design paradigms applied to implement a multi-robot system with ROS. The primary difference is the number of master nodes in the system, with centralized systems utilizing only one master node while decentralized systems more than one. A strong emphasis is placed on the difference between centralized and decentralized systems due to their respective security challenges' uniqueness. The decision to split the design paradigm into two is based on the task; centralized systems are more efficient while decentralized systems more reliable. While this decision is up to the user, it changes the system's security view as a consequence. *ROS-Immunity* is designed to address these two paradigms separately to provide a holistic security system.

Centralized ROS systems include all systems with only one ROS master node for all robots in the system. As shown in Figure 2.4, a single master node governs all communications. For users interacting with the system, the master node is the singular point of contact. All logs and priority alerts will go to the master for communication to the user. Robots interacting with the system are assumed to communicate with the centralized ROS master node.

ROS-Immunity was built on the assumptions that in a centralized system:

- There is only one master node for all robots in the system.
- All robots are aware of the master node, and it is the only ROS master with whom they can communicate.
- Robots must communicate with the ROS master for functionality.
- All robots must have a direct, non-proxy connection with the master node.

Furthermore, it is assumed that the master node can either reset and restore other robots or, at the very least, place them into a fail-safe state. This is a safe assumption because many safety regulations require both a hardware shut-off and a single centralized point for system shut-off.

The goal of *ROS-Immunity* for centralized systems is to ensure that the master node is always protected and aware of every robot's underlying status in the system. The aim is to quickly identify a compromised robot to restore them to a functioning state or a fail-safe state until the user can address the problem. *ROS-Immunity* is tested on two centralized systems: a self-driving car and a centralized factory.

Decentralized systems are those where there are multiple master nodes with a lack of hierarchy. This lack of hierarchy could be due to an impossible-to-establish or impossible-to-verify hierarchy. This includes any system where there are two co-equal masters for different components or any system in which parts of a robot cannot communicate with the absolute master and rely on an intermediary. As illustrated in Figure 2.5, in such a system, robots may communicate with multiple master nodes, but no master node holds a higher status than all the others. While ROS2 has direct support for such systems, ROS1 instead relies on packages and other workarounds. Thus, a decentralized ROS1 system cannot easily apply any cryptographic key-based system.

When dealing with decentralized ROS systems, the core assumption is that no robot holds absolute authority within the system. This may include systems designed not to have one central robot or systems designed to have one central robot where the robot cannot reliably communicate its authority to all others. In decentralized systems, robots have to deal with Byzantine fault tolerance problems, and that they must rely on consensus algorithms to detect misbehaving robots in their midst. It is assumed that every single robot in a decentralized system can communicate with at least two others and that regardless of which robot fails, there is still a path to route traffic on the network. If the latter is false, this solution is still valid; however, some targets become far more critical than others. While regulations require a hardware shut-off for decentralized systems, software shut-offs are far less likely. As a single compromised robot could theoretically power off all of the other robots it communicates

with, it is not assumed that any part of the system can force the rest into a fail-safe state. However, if an issue is detected, misbehaving robots can be quarantined, and users can be notified to activate the physical shut-off.

The main concern with decentralized ROS systems is that it is complicated to apply any cryptographic key-based system. ROS1 relies on packages and workarounds and does not directly support these systems, while ROS2 includes limited support.

The goal of *ROS-Immunity* with decentralized ROS systems is to limit the damage that any single robot can do and maintain an accurate record of potential malfeasance on each robot's part. The aim is to provide a solution that a user can use to quickly determine the source of issues for efficient remediation. As the most common implementation of decentralized systems occur in swarm robotics and in systems where there is unreliable networking, the effectiveness of *ROS-Immunity* is analyzed on a robot swarm and in a decentralized factory setting.

4.3 Overall Architecture

In this section, the threat model used to create *ROS-Immunity* is discussed. Further, the architecture designed to address the two ROS system designs in the development of *ROS-Immunity* is shown.

4.3.1 Threat Model

For the ROS system's threat model, it is first assumed a user has implemented a cryptographic communication system (either Secure ROS or SROS for ROS1, or the native secure DDS for ROS2). While this is not the norm as far as ROS system design is concerned, there is a strict limit to what users can do to secure their ROS systems without a cryptographic system. Previous research has fully mapped out this limit [165][164][137]. No assumptions are made about the security of any ROS packages or a user's ability to patch the system's underlying software.

As far as attackers go, it is assumed attackers can access the network of a robotic system, communicate arbitrarily with any component, and discover arbitrary vulnerabilities. An important exception to this is the master of a centralized ROS system, assumed to have

a minimal attack surface. Arbitrary code execution on the master of the centralized ROS system is a threat, but no current solutions exist to address it.

Further, it is assumed that while identical robots can be compromised with identical vulnerabilities, an attacker can only compromise one robot at a time initially. After the first robot is compromised, it is assumed that attackers can attack any other robot, including any robots that share a connection with any compromised robots. It is thought that the attacker has no way of disabling the hardware safety switch required by most robotic safety standards. If there is a software safety switch, it is assumed that an attacker can only disable it for robots that they fully control.

There is no assumption of any shared secrets or other requirements on the attacker's behalf, essentially assuming that the attacker has a full white-box of the system and the underlying robots. Finally, it is thought that if the system can successfully fingerprint the attacker (such as their behavior, source, or entry point), it is possible to end an attack on the system.

It is worth emphasizing the ROS Bridge[71] family of nodes, a system built to allow communication to a robotic system over the internet. This family of nodes tends to function by taking a JSON message of the topic and value that a user wishes to publish and translating it onto the ROS system. As communication over the internet introduces a different set of vulnerabilities, this family may be the most vulnerable point of attack. For this threat model, there is no assumption that the tool can protect ROS Bridge from malicious JSON messages, only that it can prevent the publication of exploits.

The solution was developed considering the threat model with the following restrictions.

- As it is assumed that a given vulnerability may be difficult to patch, the only remediation available is to prevent the vulnerable robot from receiving messages that contain the exploit.
- As it is assumed that an attacker can fully compromise any arbitrary robot, except the centralized master, any given robot can exhibit 'bad' behavior at any point in time.
- Previous trust is not evidence of current reliability.

The final point is a crucial aspect of *ROS-Immunity*. Although all communication in a system is encrypted initially, it is assumed that they can intercept traffic once an attacker begins. Thus, any algorithms used to decide new rules cannot depend on trust or shared secrets.

4.3.2 Solution

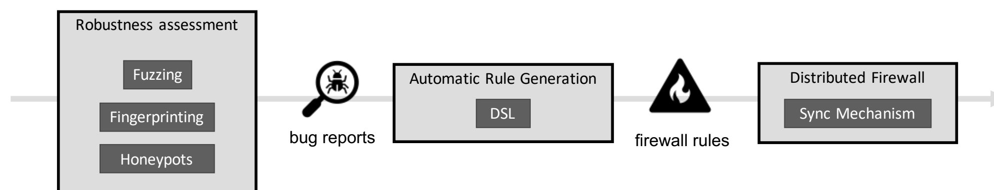


Figure 4.1: *ROS-Immunity*'s three components to implement an integrated security mechanism in ROS system addressing the security gaps in current ROS systems.

ROS-Immunity implements an integrated security mechanism in ROS systems, addressing the security gaps in both ROS1 and ROS2. *ROS-Immunity* consists of three components, as shown in Figure4.1: robustness assessment, automatic rule generation, and a distributed firewall.

The robustness assessment component integrates several ROS security tools to help the user identify vulnerabilities before an attacker can. Additionally, it should help identify if an attacker has compromised any of their robots or systems. Once bugs are detected, an automatic tool reads their reports and generates rules in a domain-specific language. Additionally, the robot administrator can independently write their own rules based on other external sources of bug reports. An integrated firewall is deployed to protect a single robot by filtering incoming connections for malicious packets or other 'bad' behaviors. It intercepts packets before entering the ROS system and checks them against a known fingerprinting database for potential vulnerabilities. A synchronization mechanism addresses the challenges of multiple robots in both decentralized and centralized environments. They act to disseminate rules to all robots and alert the user of suspected compromised robots. The next sections present the three components in detail.

4.4 Robustness Assessment

As a cyber-physical system, ROS not only inherits security vulnerabilities from both software and hardware but is also open to a new class of vulnerabilities from interactions with the real-world. Cyber-physical systems are known for software and hardware issues, such as weak cryptography, inadequate protection mechanisms, sensor vulnerabilities, and have severe resource constraints for addressing these issues. All ROS systems suffer from these vulnerabilities and interactions to the real world, exposing them to additional vulnerabilities with a high cost: damage from any robotic system malfunctions on its surroundings or itself. The most significant example of these vulnerabilities is the StuxNet worm of 2012, where outside intervention led to the destruction of dozens of centrifuges. Similarly, within the robotics world, a similar analysis of the RAVEN II surgery robot revealed critical security vulnerabilities enabling attackers to execute arbitrary commands[41]. Therefore, fully accounting for all three levels of vulnerabilities is of vital importance.

To account for all vulnerabilities that could affect a ROS system, a fully integrated tool is needed. The most critical component of a security tool is the ability to discover new vulnerabilities in a system. There have been several previous research endeavors to detect vulnerabilities for ROS systems, including static code analysis [112][62], property-based testing [125], Honeypots[78][29] , and fuzzing[187][128]. All of these disjointed techniques yield valuable insights into the security of a robotic system, and as such, any integrated security system must make full use of them. It is critical to make sense of what information each method provides to combine them into a complete picture.

Alone, each of these solutions can provide coverage for only a proportion of these vulnerabilities. In particular, Honeypots are very adept at finding currently utilized real-world exploits but are slow to identify new anomalies and require continual upkeep to ensure attackers do not detect them. Meanwhile, static code analysis can be done before a system is released to help ensure the quality of the software; however, it is prone to false positives and requires significant developer attention. Table 4.1 lists the strengths and weaknesses of these systems when used alone. Any system cannot address the complex security ecosys-

tem surrounding ROS systems.

Type	Strengths	Weaknesses
Static Code Analysis[112][62]	Fast and easy to use	Large amounts of false positives
Honeypot[78][29]	Detects real world vulnerabilities	Requires constant maintenance
Fuzzing [187][128]	Finds runtime issues not seen by other models	Performance heavy and programmatically difficult
Property Based Testing[125]	Leverages existing test infrastructure	Requires manual intervention to analyze bugs

Table 4.1: Strengths and Weaknesses of known ROS security tools

The need for uniformity further complicates this. Once a developer has chosen which combinations of tools they want to employ, they must use a fingerprinting standard. Without this standard, it is difficult to secure a ROS system once a vulnerability is known. For example, the outputs of a Honeypot and a static code analysis tool are very different, the former being raw data and a log of issues, and the latter reporting an error message specifying problematic lines of codes. These outputs are not initially compatible and must be combined in a way that can be utilized by the system. A standardized way to approach this is to export all results into the ROS security framework (RSF), where vulnerabilities are described in terms of the effected sub-system, the effect of the attack and known triggering methods [128]. Methods such as these do exist, such as the development of describing robotic vulnerabilities similar to the MITIR CVE[13]. Once a developer attains all fingerprints, vulnerability analysis and vulnerability mitigation (until a patch has been created) are needed. In order to check for the presence of known vulnerabilities, automated scanning tools such as *ROSploit*and ROSPenTO are used.

For this dissertation, a combination of anomaly detection, Honeypots, and fuzzing are utilized to detect vulnerabilities in robotic environments. All events are reduced to a common fingerprinting language. Fingerprints are constructed by identifying the source topic or group of topics affected by the vulnerability. It also identified a minimal encoded string required to recreate the vulnerability. This string can be a list to encode sequential information.

To distribute the fingerprint on a centralized system, it is passed from the master to the

designated automatic rule generation node. On the contrary, in a decentralized system, there is no singular trusted rule generation node. Therefore, alternatively, any robot may propose a new rule by providing a fingerprint. Other robots will vote on the new rule by emulating the relevant part of their system in a sandbox. This voting system ensures that attackers cannot perform a denial-of-service attack on the system by proposing new rules.

Fingerprinting in a decentralized system poses a challenge: allowing a user to distribute a large batch of rules at once can cause a delay. As each rule has to be individually confirmed and verified, this process may take a significant amount of time. However, if a user sets up their system to access a simple majority of robots, rules can be deployed as generated rules rather than fingerprints and will propagate readily throughout the network. In this case, users must ensure a majority, or all rules will be denied and assumed to be an attack. While this adds a layer of complication, it is demonstrated that this method cannot be compromised, as the distributed firewall will identify and stop attacks.

In both cases, these fingerprints are given to the automatic rule generation system and harden the environments.

4.5 Automatic rule generation & Distribution

Once a developer has successfully fingerprinted all known vulnerabilities in their ROS system, it is critical to have an efficient way to translate those fingerprints into actionable firewall rules. These rules are the framework utilized to develop patches for vulnerable ROS nodes and may include node communication constraints or packet blocking for particular values.

An automatic rule generator was developed to take vulnerability fingerprints and translate them into rules. This generator takes a fingerprint, in the form of a topic, node, field, or value, with the ability to provide wild-cards (at any level, except topic), and begins generating rules in a Domain Specific Language (DSL). It generates several hypothetical rules as specified by the user. It uses the fingerprint as an initial seed to optimize the generated rule, producing the most minimal effect that fully encompasses the vulnerability fingerprint. In this way, the tool does not accidentally block any packets that are not vulnerabilities and ensures that each rule is narrowly-targeted. This automatic rule generation allows a user to maintain

a list of fingerprints, subscribe to an existing list, and avoid time-consuming manual rule generation. All rules are then given to the firewall for implementation, discussed in Section 4.6.

Rules are composed of three key components: behaviors, affected systems, and values. The first, behaviors, encompass the part of the robot that needs monitoring, such as the network interface, file system, or processor behaviors. Affected systems are the ROS-level or network-level identifiers to apply the rule to, such as: specific robots, IP information, topics, nodes, and parameters in parameter server. Values are the specified values the firewall should monitor, such as string length or the presence of invalid messages. A single rule can apply to one or a combination of these components.

The automatic rule generator's ultimate goal is to produce rules that block the vulnerability described in the fingerprint without interfering with legitimate ROS traffic. To optimally generate, the system begins by compiling a simple rule that purely filters all traffic matching the fingerprint. Once compiled, it performs a second check to determine edge-cases that can cause the rule to fail. This second check is conducted by generating messages that comply with a given rule, and checking that it also complies with the fingerprint. It recurses on the generated rules until it reaches a final, edge-caseless rule.

Occasionally, erroneous states may be hard to fingerprint, such as high-network traffic from a denial-of-service attack or changes in publication timing from a man-in-the-middle attack. In these cases, it may be of interest to manually define rules. Using the DSL, a user may manually define rules to filter specific behaviors in ROS nodes, services, and topics, to be passed to the firewall. Even manually, this method allows for rapid rule generation in a way that is not yet available, providing a user with great flexibility and a simple system for rule generation.

Both of these methods provide simple, rapid methods for rule creation. As of 2020, there are no known systems that provide a way to filter ROS traffic using ROS syntax that persists through relaunches of the robot. A user must manually identify and specify the port and connections for every node and topic to define rules, which may change whenever a ROS system is restarted. This is often unrealistic and leaves users without the ability to apply

rules to ROS traffic. It is important to note that as most firewall rules filter any packet that matches, overly broad rules may damage system operation, as in any rule-based system. An optional mitigation feature in a decentralized system is to compare percentages of matched traffic before and after new rule integration. With this feature, each robot may store a small snippet of previous traffic before a rule is updated and check the percentage of traffic that matches the new rule.

Once the system has a collection of rules, it needs to distribute them among the connected robots. To begin, it organizes the rules in a tree-based structure based on which topic or node to which the rule applies. The automatic rule generator attempts to develop trees such that any rules for nodes that share topics are close together. Individual firewalls may reconstruct this tree to serve their particular purpose best. Once this tree has been assembled, it is hashed, and that hash is taken as the current reference hash.

In centralized systems, the tree is passed to every single robot, and each robot is responsible for compiling the rules into their firewall. They must respond with the hash of their current firewall rules to confirm that the distribution was a success.

In decentralized systems, any robot may propose a new tree containing any number of rules. However, they must offer some *credibility* through the Tendermint protocol. This *credibility* is wagered against the other robots adopting the rule, such that if the rule is not accepted, the offering robot is unable to propose new rules for a set time and is monitored closely for other malicious behavior. Robots regularly receive *credibility* from their neighborhood for 'good' behaviors, i.e., those that do not trigger an alert with the firewall.

Another critical component required is the time synchronization of the various firewall components. As the firewalls need to know when a connection has been initiated with a ROS master, the system needs to enforce a consistent time to enforce rules properly. On a centralized system, this is very simple, as basic ROS functionality currently enforces it. Additionally, a centralized master can act as an NTP server. However, for decentralized systems, each robot has to maintain its GTP server, adding a layer of complication [12]. When two or more robots exchange rules, a GTP handshake is conducted. If a rule affects communication between robots, it is flagged to trigger a time synchronization check. To minimize

the amount of GTP synchronizations between robots, rules that do not affect communication between robots, such as rules that only apply to one robot, do not require this handshake or synchronization. Time synchronization also occurs during rule distribution, discussed in Section 4.6.

4.6 Distributed Firewall

One of the ROS ecosystem's greatest strengths is the modular nature of its packages, allowing for rapid software installation. Unfortunately, this carries with it a security weakness: the majority of code running on any given robot tends not to be developed by one particular developer[141]. This can lead to patching and vulnerability mitigation issues as each developer must either maintain separate branches of standard ROS modules or pardon their system while waiting for the open-source community to address vulnerabilities. Currently, the community is very active, but as robotics move into more domains of life, there will be a greater need for highly responsive vulnerability solutions.

In conjunction with an adequate ROS cryptography system, a ROS-aware firewall is the best solution to mitigate this problem. By allowing robots to continue running in a secure state until the community has had ample opportunity to develop a patch, developers simultaneously increase system security and avoid choosing between costly shut-downs and substantial risks caused by vulnerabilities. The *ROS-Immunity* firewall system is developed to block unauthorized access to topics and service connections and filter out any messages with identified vulnerabilities until the system is patched.

A ROS firewall would need to be aware of both ROS middleware syntax and underlying network connections in theory. It would need to monitor every message published for known vulnerabilities and help prevent attackers from using compromised robots to destroy the system. Additionally, it would need to protect against denial-of-service and other resource exhaustion attacks. A small processing and memory footprint would be required to ensure any robotic system's proper functioning, as small changes can significantly impact such systems. Theoretically, this may be conducted using access control enforcement and filtering suspected compromised packets, but, as of 2020, no known system has yet to be

developed.

As with all theoretical solutions, there are several challenges for developing such a system, especially when considering multiple robots across different processing systems, a common problem in the ROS ecosystems. Providing enforcement with topics running on different networks than their masters is considerably more challenging than topics and masters running on the same system. Users would need to ensure that the robot correctly asked the master node for the topic's location and did not utilize a scanner to find it. Additionally, one would have to ensure that each processor runs the same version of the firewall to provide consistent security and ensure that filtering is done on all systems simultaneously while operating under the same performance and memory constraints.

In this section, implementing a distributed ROS firewall that addresses all of these issues is outlined for both centralized and decentralized robotic architectures. This solution provides an efficient way to take novel vulnerabilities and translate them into patches and detect when a robot has been compromised and mitigate damage to other robots in the network.

The firewall is implemented with eBPF/XDP with modular drop-in support for a byzantine fault-tolerant algorithm. eBPF is an extension of the Linux Berkley Packet Filter that allows users to execute unique code in kernel space[74]. XDP is an extension to eBPF, allowing users to bypass the traditional Linux network stack for efficient packet processing[66]. Tendermint[36] is used as the Byzantine fault-tolerant protocol. It supports both centralized and decentralized ROS systems and has a simple interface for alerting users of potential vulnerabilities. The centralized solution includes one keystone master firewall that all robots share, and the decentralized solution requires each robot to maintain its master firewall.

Figure 4.2 shows the architecture of the distributed ROS firewall, highlighting the differences between a centralized and decentralized robot. The firewall is a collection of individual robot firewalls and a standardized interface to communicate between them. In the centralized firewall, this takes the form of a master firewall that acts as the centralized repository for all known rules. All other firewalls must report to the master firewall to keep the central ROSgraph up-to-date. As the ROSgraph is distributed across multiple robots, all robots must continuously enforce ROSgraph altering behavior, such as creating new nodes, subscribing

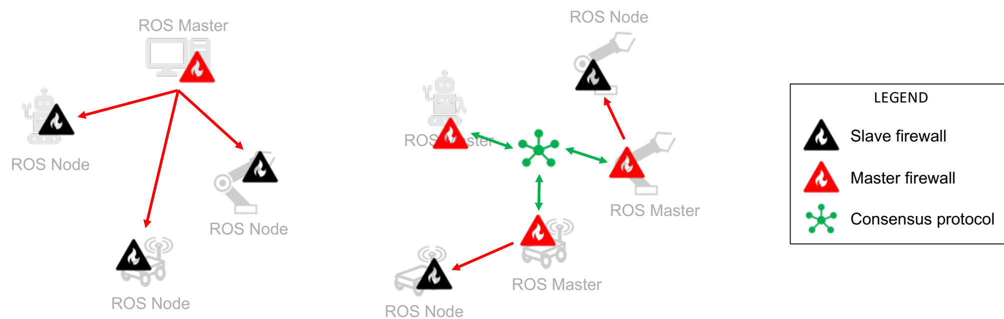


Figure 4.2: Integrated Firewall Architecture for centralized (on the left) and decentralized (on the right) ROS system designs.

to topics, and launching new services.

Meanwhile, in a decentralized firewall, each robot makes a cluster with their n user-specified nearest neighbors with whom they share rules. For a rule to propagate through the system, the Byzantine fault-tolerant Tendermint protocol allows robots to propose new rules and other robots to vote based on a distributed verification system. This verification system works as follows: a robot can individually verify a proposed rule if they have adequate processing power. Otherwise, a consensus algorithm is employed where all robots with spare processing power vote to decide inclusion. All firewalls must have the same rules to ensure that a user can patch any given node or topic. The firewall stores information in terms of topics, nodes, and services and automatically translates them into ports, IP addresses, and values. Then, this information is filtered by the firewall with the specified rules. Another benefit that this system provides, in this case, is that as robots maintain a network of other robots' information: once one robot begins acting strangely, other robots are aware of the strange behavior. This allows the decentralized system to adapt to an attacker dynamically: once a robot is compromised, other robots can react by filtering traffic away from it.

The process of applying new rules differs substantially based on the type of the system. In the centralized system, new rules can be added by the user to the master firewall. The master firewall will schedule a patch to every single robot in the system in parallel, using the number of rules added as the differentiator for time. Then, the system and master exchange keys for validation to prevent master impersonation. The master transmits the

new rules, which the system integrates and compiles a hash of the complete rule-tree. If that hash matches the master's, the master moves onto the next robot; if not, it asks for formal verification. The robot enters a safe standby mode until verification is complete.

For decentralized systems, this process is more complicated. The user can choose to upload the rule manually to most of each neighborhood's robots, who will then propagate it to the others, or launch a collection of rule-adding robots. The first method is more efficient if there is a reliable connection between the rule system and the robots, as newly launched robots will have very low *credibility* initially. The second method is more efficient if the distributed system does not have a clean, consistent, easy way to access the network, such as in the case of mesh and dynamic swarm networks. Both methods follow the same path of gradual rule propagation. However, instead of verifying with a hash of the rules, robots have to share a signed timestamp when they last updated their rules to ensure all robots are updated to the newest version. An entire neighborhood would have to be compromised to push alternative rules. Otherwise, the consensus algorithm will identify and exclude compromised robots.

As discussed previously, a key component of system monitoring in both centralized and decentralized systems is time synchronization. In a centralized system, the master periodically updates each robot with the current time. The firewall raises alerts if any packets are translated with an incorrect time, preventing attackers from communicating on topics. In a decentralized system, only inter-robot communication is constrained with packet time-monitoring. Communicating systems sync their clocks at a rate determined by the user. If there is a notable drift in time between packets on an inter-robot topic, all subscribers notify the rest of the system of the potential anomaly.

The firewall vastly improves the security of ROS systems in two ways: 1) it prevents any ROSgraph-based alterations that are not approved by the master; and 2) it provides packet-level filtering that works with any ROS encryption mechanism, such as SROS or Secure ROS, to patch known vulnerabilities. In the first case, the firewall prevents any connections that do not initially communicate with the ROS master for connections and port openings. This is assumed to be a vulnerability as the master has no formal way of vetting the under-

lying system, and any break between the masters' model and the underlying system allows attackers to inflict severe damage. In the second case, packet-level filtering blocks any packets that match a fingerprint, as generated by the automatic rule generator discussed in Section 4.5, allowing developers to patch known vulnerabilities without changing the underlying code rapidly.

As with all security systems, this firewall introduces a small risk of an attacker using the system to introduce a denial-of-service attack through the advent of malicious rules. In this case, if an attacker controls the master node for a centralized system or more than a third of a target's neighborhood's worth of individual robots in a decentralized system, the attacker can publish false rules that disable critical communication between systems. This could go as far as denying all communication between nodes with topics, completely halting the system. While the firewall provides considerable protection against this, if such a scenario does occur, the only remediation is for the owner to repair the rules and restore the compromised robots manually. This is not significantly different from the standard remediation for other similar attacks. However, given the structure of ROS systems in general, any attacker who could successfully introduce this type of attack with *ROS-Immunity* could inflict far more damage without it. With the security additions provided by the use of the firewall, the benefits far outweigh this risk.

4.7 Use Cases & Experimental Evaluation

To test the system, four separate common ROS use-cases covering the most common real-world robotic situations were designed. For each use-case, nine configurations of nodes and rules were analyzed. Both robot count and rule count were varied to examine the cost of *ROS-Immunity*. Each use-case was simulated using Docker[37] containers for each robotic system, linking them on a virtual network with the ability to monitor all incoming and outgoing connections. These containers contained their version of *ROS-Immunity* and the installation of the firewall. A ROSBag file containing the test data was partitioned onto each container such that each simulated robot could only access data their sensors could plausibly see.

The first use-case is a ROS self-driving car with a cluster of specialized nodes commu-

nicating over a TCP network. This car is modeled after the Autoware[44][110] a self-driving car with various features enabled to test different levels of complexity. The most simplistic features include a self-driving car with two sensor nodes(LiDAR, lane camera), a planning node, a control node, and a processing node. This is the configuration used in the 5-node experiment. The 10-node configuration adds a traffic camera and ultrasonic proximity sensor, and the 15-node experiment adds GPS and a pedestrian tracker, and a back-up control node in case the first fails. All iterations of this experiment are considered a centralized system.

The next use-case is based on an example from swarm robotics. Consider a case where there are many small self-contained robots with minimal features or processing power. Each robot in the swarm forms a group with its neighbors, and data is shared with only their neighbors, not the entire system. A swarm is given a simple goal of identifying and 'capturing' flags. The run concludes when every flag has been 'captured,' and overhead is measured. The swarm is a decentralized system of robots, where each robot is an identical copy, and the entire system shares a goal.

The final use-case is a ROS factory with robots working along an assembly line. Two different architectures are evaluated for this use-case: a centralized architecture based on the ABB pick-and-place example and a decentralized architecture based on a real-world recycling plant. For both configurations, robots were tested in groups of 10, 20, and 50. The centralized architecture is designed so that all sensors pass data for all systems to a central processing server, which then issues commands for each robotic arm. The decentralized architecture is designed so that each robot is acting as a self-contained detection system for a type of recycling system. Each robot passes along only the deltas of what their cameras see to robots down the line, and these neighbors utilize that information to develop the most optimal behavior for when materials reach them.

This use-case was split by type of system as they are both common infrastructures with unique issues, allowing a direct comparison of the centralized and decentralized approaches. Both architectures were scaled by adding additional assembly lines in parallel, with an additional assembly line added for each set of 10 nodes. For example, a 10-robot

system has one assembly line, and a 50-robot system has five. This is added due to the decentralized system's design: each robot in the assembly line is focused on one recyclable. For consistency, it is reflected in the centralized system as well. To ensure the simulations are accurate, it was confirmed that the setup had similar behavior to a real-world ROS installation.

Use Cases	Separate Processors	Cross-processor topics
Car	5	25
Car	10	30
Car	15	35
Swarm	20	5
Swarm	50	10
Swarm	100	21
Centralized Factory	10	7
Centralized Factory	20	14
Centralized Factory	50	25
Decentralized Factory	10	5
Decentralized Factory	20	5
Decentralized Factory	50	5

Table 4.2: Breakdown of the use-cases and experimental evaluation of *ROS-Immunity*

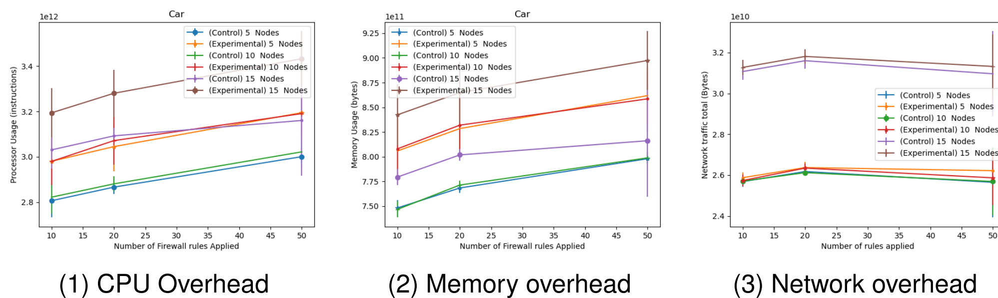
To evaluate the use-cases, bag files were constructed with a randomized mixture of regular data and added attacks. Both attacks had firewall rules as well as attacks that did not were analyzed. For attacks that were known, it was confirmed that the firewall successfully stopped them in all cases, and then the overhead and false-positive rate were measured. For unknown attacks on a decentralized system, the length of time between the attack and when the anomaly detector reacted and excluded the misbehaving robots, and when the automatic rule generator created a rule and the firewall stopped the attack.

Multiple trials were conducted within each use-case where the number of security rules added to the system and the number of robots working in concert varied. For each of the four use-cases, three configurations of robots were compared. For each configuration of robots, three configurations of rules were considered: a count of 10, 20, or 50. Then, each trial was repeated 16 times to get a better representation of how the system performs. In

total, 576 experiments were performed.

Furthermore, to test the firewall functionality, 50 vulnerabilities were generated for each use-case. These were generated from a mix of well-known ROS vulnerabilities, previously patched vulnerabilities, and prior security research. These vulnerabilities were introduced into the test environment, and firewall rules were added to protect the robot. For each of the 12 use-case and robot configurations, vulnerabilities were injected into the control bag-file in groups of 10, 20, and 50 rules to test the impact of different rule configurations on the system. Once added, the control bag-files were modified to contain vulnerabilities corresponding to the chosen number of operating rules (10, 20, or 50). The rules were then loaded onto the system to confirm that the system continued to function despite the vulnerabilities. For each trial, a random grouping of vulnerabilities was selected and placed randomly throughout the bag-file.

The experiments begin with a fingerprint of a known vulnerability. A collection of these fingerprints is passed to the automatic rule generator, which develops rules to account for them. These rules are loaded onto the simulated firewalls. After loading, data with the vulnerabilities is passed to the systems. The systems execute normal behaviors. Once the simulation is complete, the CPU, memory, and network overhead are measured, and final results are averaged over the 16 trials. False-positives were detected by exploring discrepancies in the network data.



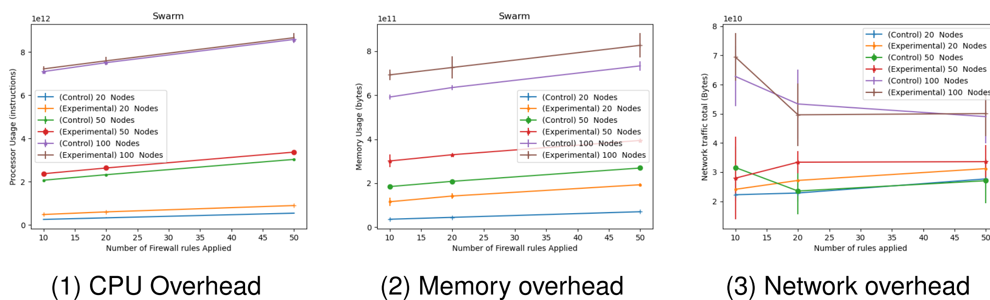
(1) CPU Overhead (2) Memory overhead (3) Network overhead

Figure 4.3: *ROS-Immunity*: Results of Experimentation on Self-Driving Car

The results of the experimentation are summarized in Table 4.3, as well as Figures 4.3, 4.4 4.5, and 4.6. Overall, the performance of *ROS-Immunity* was auspicious, producing

Use-case	Rule count	False positive rate
Car	10	0-4%
Car	20	6-10%
Car	50	8-14%
Swarm	10	0-8%
Swarm	20	0-8%
Swarm	50	4-12%
Centralized Factory	10	0-5%
Centralized Factory	20	6-11%
Centralized Factory	50	4-17%
Decentralized Factory	10	0-2%
Decentralized Factory	20	4-6%
Decentralized Factory	50	4-12%

Table 4.3: *ROS-Immunity*: False positive rate per trial

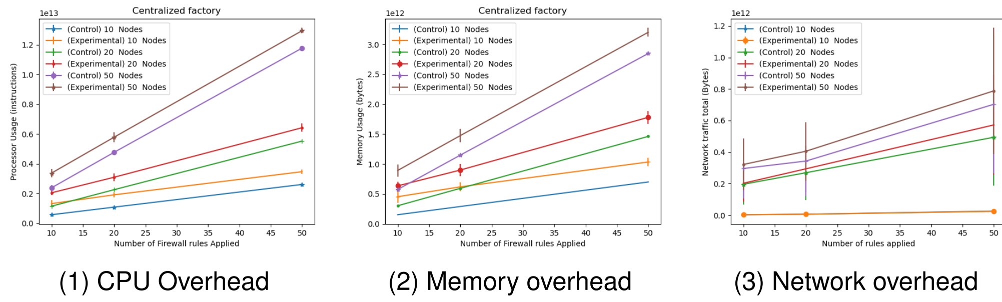


(1) CPU Overhead (2) Memory overhead (3) Network overhead

Figure 4.4: *ROS-Immunity*: Results of Experimentation on Swarm

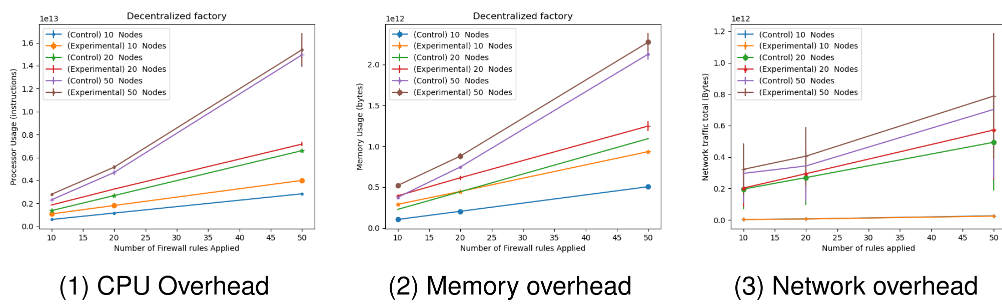
small processor and memory overhead in all use-cases. It was discovered that the decentralized implementation scaled well with the number of robots, while the centralized was far more efficient with smaller systems. The in-line firewall filtering stopped 100% of known attacks with a worst-case 17% false-positive rate.

Table 4.3 displays the false-positive rates for each of the use-cases. For each use-case, false-positive rates are calculated by the number of rules used in each trial, 10, 20, or 50. The lower bound of the false-positive rate is the number of rules that incorrectly filter out a packet, compared to the total number of rules run in the trial. The upper bound is a conservative estimate that treats every single trial that has at least one rule that triggers a



(1) CPU Overhead (2) Memory overhead (3) Network overhead

Figure 4.5: *ROS-Immunity*: Results of Experimentation on Centralized Factory



(1) CPU Overhead (2) Memory overhead (3) Network overhead

Figure 4.6: *ROS-Immunity*: Results of Experimentation on Decentralized Factory

false-positive as a false-positive.

The worst-case scenario was considered to calculate the false-positive rate for the upper bound. For each trial, the three different experimental setups were combined with a different number of robots with the same number of rules. Any scenarios where a packet was filtered was identified. If any packet[3] was filtered as a vulnerability from the system, the entire trial had a false-positive in at least one robot. Therefore, the false-positive rate was calculated as the worst-case scenario: even if only one rule was triggered, the entire trial was considered a false-positive. As these are safety-critical systems wherein filtering out vital information from the robot when it is not part of an attack is an unacceptable risk. Thus, the reported are very cautious false-positive rates.

Overall, the false-positive rate is very low for all use-cases. The false-positive rate was higher on systems that relied on video data and vulnerabilities related to images, an ex-

[3] Packets that were added for testing were excluded.

pected result. While typically 10% is considered a high false-positive rate, in this case it is the absolute worst-case, and the actual false-positive rate is likely much lower. However, there is an opportunity for future improvement of the false-positive rate through a more intelligent rule design to reduce false-positives.

The car has an average CPU overhead of 7% ($\sigma = 4\%$) for 5 nodes, and 5% ($\sigma = 3\%$) for both 10 nodes and 15 nodes. It has an average memory overhead of 8% ($\sigma = 5\%$) and an average network overhead of 2% ($\sigma = 1.5\%$) across all three node layouts.

The self-driving car results were very consistent and positive, illustrating the superior performance of *ROS-Immunity* on centralized systems. In this case, as the firewall filters large amounts of video traffic and other data-intensive messages, which would typically produce significant overhead, the benefits are readily apparent. Growth in network bandwidth by the number of rules can be observed due to the generation of bag-files. With the generation of bag-files, for every vulnerability added, new data was added for both control and experimental to load the vulnerability into the experimental bag-file correctly. The addition of new data was conducted in both control and experimental for consistency, but no vulnerabilities were added to the control.

The swarm has an average CPU overhead of 70% ($\sigma = 5\%$) for 20 robots, 10% ($\sigma =$ %1) for 50 robots, and 1% ($\sigma = 1\%$) for 100 robots. It has an average memory overhead of 150% ($\sigma = 20\%$) for 20 robots, 50% ($\sigma = 5\%$) for 50 robots, and 10% ($\sigma = 2\%$) for 100 robots. It has an average network overhead of 10% ($\sigma = 1\%$) for 20 robots, 15% ($\sigma =$ (2%) for 50 robots, and 27% ($\sigma = 3\%$) for 100 robots.

There is a distinct trade-off between smaller and larger swarms. Smaller swarms require more CPU and memory usage to run the consensus algorithm and firewall, producing a higher overhead. On the contrary, larger swarms are very computationally efficient but have a higher network overhead to handle all the communication between systems.

Meanwhile, network overhead displays some strange behaviors due to swarm behavior. For the swarm bag-file, depending on the number of vulnerabilities added, there was a potential configuration wherein the act of attempting to set a bounding box on distance would cause the swarm to have to manually re-position and recreate the network. As the experi-

ment runs, the network shape is non-deterministic, and each reconstruction requires a large amount of bandwidth usage. This leads to higher variance in the network usage between runs and many inconsistent results.

The centralized factory has an average CPU overhead of 8% (σ = 3%) for 10 robots, 15% (σ =6%) for 20 robots, and 20% (σ = 5%) for 50 robots. It has an average memory overhead of 30% (σ = 4%) for 10 robots, 14% (σ = 5%) for 20 robots and 11% (σ = 7 %) for 50 robots. It has an average network overhead of 2% (σ = 1%) for 10 robots, 5% (σ = 4%) for 20 robots, and 20% (σ = 6%) for 50 robots.

The centralized factory results showed a gradual worsening of performance as the number of robots increased for both CPU and network overhead, while memory overhead improved as more systems came online. The single assembly line has fantastic performance, while the 50 robot assembly line has higher overhead and is still within manageable limits. The rapid growth in network traffic likely comes from each new assembly line having to communicate camera data to the central planning server. This is easiest to see as the number of rules increases creating a more extensive test set.

The decentralized factory has an average CPU overhead of 10% (σ = 6%) for 10 robots, 11% (σ = 3%) for 20 robots, and 6% (σ = 1%) for 50 robots. It has an average memory overhead of 45% (σ = 4%) for 10 robots, 16% (σ = 3%) for 20 robots and 8% (σ = 4 %) for 50 robots. It has an average network overhead of 3% (σ = 2%) for 10 robots, 5% (σ = 1%) for 20 robots, and 15% (σ = 2%) for 50 robots.

The results of the decentralized factory are overall positive. The performance scales quite well with the number of robots, with very positive overhead results with high robot-count systems. As the number of robots increases, the overhead of the consensus rapidly drops and the CPU, network, and memory. Of interesting note is the clear benefits of using a decentralized architecture in both the lower network and memory usage since robots do not have to share all of their data. However, the drawback is higher CPU usage caused by each system running a computer vision algorithm.

The factories present an interesting point of comparison. Although the two factory systems are performing similar tasks, their respective overhead results are inverted. The cen-

tralized factory improves when the number of robots decreases, whereas the decentralized factory improves when the number of robots increases. This is due to the overhead of setting up a decentralized consensus algorithm being relatively constant. Simultaneously, a centralized system has to set up more direct connections with each new robot. This result provides a clear design paradigm: if the system requires a large number of robots, it is better to design it as a decentralized system.

To validate *ROS-Immunity* against unknown vulnerabilities, tests were conducted on both the time the decentralized systems took to exclude a compromised system and the time it took for the centralized system to issue a reset. The car was able to react within .52 seconds ($\mu = .41$, $\sigma = 0.09$ seconds) in the worst case, while the centralized factory was able to react within 1.31 seconds ($\mu = 1.16$, $\sigma = .29$ seconds) in the worst case. Additionally, the decentralized factory was able to react within 2.4 seconds in the worst case ($\mu = 1.71$, $\sigma = 1.39$ seconds) while the swarm took 2.1 seconds ($\mu = 1.4$, $\sigma = 1.08$ seconds). While the decentralized was slower than the centralized, it could still react and exclude a compromised system before any system connected to it could be compromised. This means that an attacker can only compromise 2+neighborhood size nodes (the initial node, its neighborhood, and one additional node) before being excluded from the network.

In Figure 4.7 the per second power requirements of *ROS-Immunity* were calculated versus the control system using the ARM embedded power formula found in Mao *et al.*[115][4]. The power draw of each robot in the system was normalized. Overall, *ROS-Immunity* had an average power cost of 7% for the car, 13% for the Decentralized Factory, 18% for the Centralized factory, and 16% for the swarm. This additional power overhead is very low compared to the power requirements inherent in hardening the network with cryptography [123].

[4]GPU power was excluded from the calculations

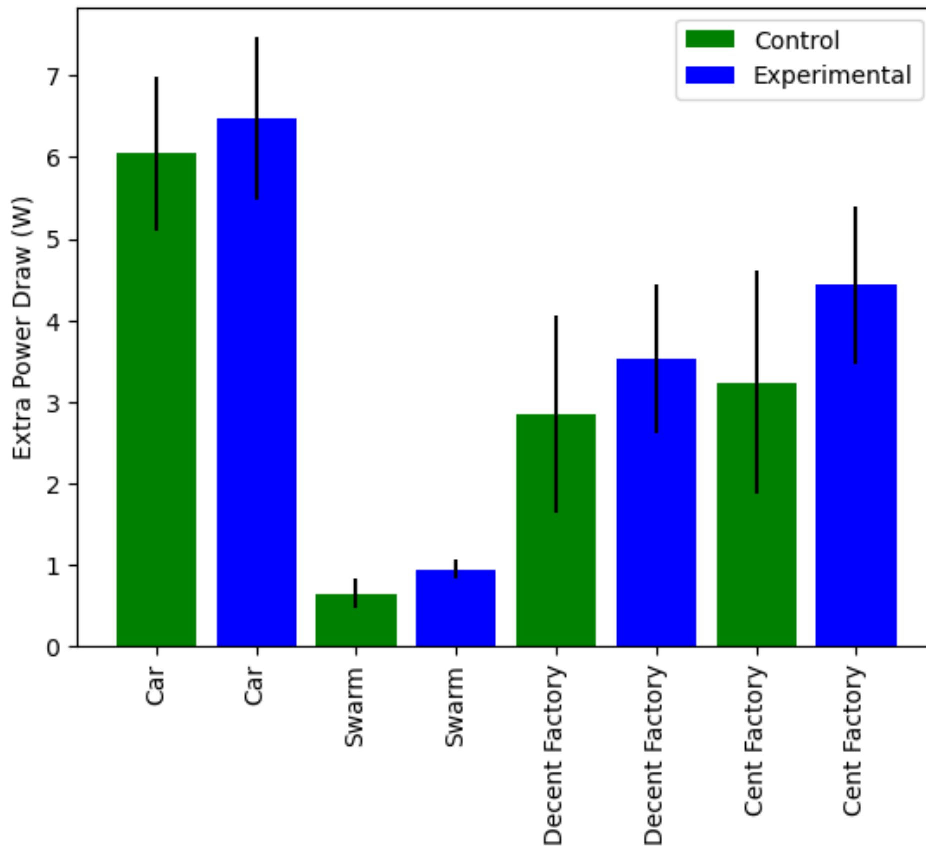


Figure 4.7: *ROS-Immunity*: Results of Experimentation: Normalized Power Overhead

4.8 Conclusion and future work

ROS-Immunity can vastly improve the security of several different robotic systems with low overhead. *ROS-Immunity* can detect all the known attacks, where the systems were loaded existing firewall rule. Further, it can detect most of the previously unknown attacks within 2.4 seconds in the worst case, 0.41 seconds for the car, 1.16 seconds for the centralized factory, 1.71 seconds for the decentralized factory, and a total of 1.4 seconds for the swarm second on average.

While *ROS-Immunity* has a worst-case false positive rate of 17% and a more typical rate of 8%, there is potential to improve this with more intelligent rule generation. In the future,

extensions could improve the false-positive rate and allow for implementation in smaller systems. It is thought that by leveraging machine learning techniques, the false positive rate can be vastly improved to design more specific rules. Additionally, we believe that there could be better implementations for small distributed systems to cut down overhead.

Part II

Internal system defense

Chapter 5

ROSDefender

"A secure system is one that does what it is supposed to."

Eugene Spafford

5.1 Introduction

This chapter introduces the primary firewall component of the integrated system described in Chapter 4, built using software-defined networking and utilizing machine learning for anomaly detection. The core principle underlying this design is that the ROS framework shifts all middleware communications up to the network layer, improving inter-operability. This creates a scenario where network-based security solutions are the most logical stepping off point to address many design issues in the framework of ROS.

The most significant goal of ROS-Defender was to address issues with the ROS master. Primarily, the ROS master's inability to prove or verify the internal ROS graph's accuracy, i.e., the master could not verify that nodes on its internal ROS graph were actively running on a system, was of interest. Second, ROS-Defender aims to detect and mitigate anomalous node behavior to address the gap in security software research. Even in the presence of cryptographic ROS solutions, misbehaving nodes can still perform attacks, such as data corruption or denial of service attacks, which can compromise a system. Current solutions for ROS security do not protect compromised nodes or denial of service, do not support

reactive policies that are updated dynamically at runtime, and cannot enforce low-level granularity network policies.

Adding access control to the existing ROS codebase would require significant engineering efforts. The design choice taken by ROS-Defender is to leverage Software-Defined Networking (SDN) to monitor the communication among the nodes and enforce (if needed) access control rules. Software-Defined Networking (SDN) creates new opportunities to secure ROS at the network level by providing a framework to define, enact, and enforce per-flow policies in a dynamic manner. However, users of ROS would prefer to enable policies that follow application-semantic, i.e., have policies per *topic*. Thus, is it desirable to accommodate not only network-level but also topic-level policies and to be able to enforce such policies dynamically.

ROS-Defender secures ROS by leveraging SDN to control per-flow policies by augmenting per-flow control, with a per-topic control to accommodate ROS semantics. Two security applications on top of this SDN framework are introduced: a firewall and a monitoring-watch. Both applications communicate with the SDN network to react to observed traffic and detect either violation of existing policies, or new attacks and adjust the policies. Further, ROS-Policy-Language, a policy language for ROS applications, is defined to allow a user to express access control rules within the ROS semantics and vocabulary, which are then mapped to network-level rules that can be enforced by the SDN-based framework.

It is demonstrated that ROS-Defender can efficiently locate compromised nodes and halt their execution, as well as enforce the master's ROS graph (and all its assumptions) on the system. At the time of its development, ROS-Defender was the only tool with the ability to prevent denial of service attacks and protect the ROS graph. However, for all of its strengths, ROS-Defender has three significant drawbacks: lack of ROS2 support, performance issues, and the inability to detect data attacks.

This chapter concludes with a discussion of rosdefender's contributions to ROS-Immunity. The foundation and insights of the development of ROS-Defender built the backbone of the firewall component of ROS-Immunity. The key insights are discussed, including changes to configuration, user-interfacing, and improving parallel modularity.

5.2 Adversarial Model

In this section, adversarial models considered when designing defense mechanisms for ROS are discussed. First, the system assumptions that apply when ROS or SROS are deployed are outlined. Then, two adversarial models that match the ROS and SROS deployment, respectively, are described.

5.2.1 Independent Assumptions

The following independent assumptions are held. These assumptions concern the ROS ecosystem, regardless of which ROS features are running.

- It is assumed that the master node acts like a certificate authority (CA) or a domain controller and is the root of trust for the system. It is also assumed that the master node is not compromised before the system is started.
- It is assumed that the parameter server has protection mechanisms from malicious alteration for the parameters stored on the parameter server. A compromised node can change only parameters to which it usually has access.
- It is assumed that the ROS middleware is secure from exploits.

All ROS systems run on top of the Linux operating system. Linux is host to a set of its own security risks and vulnerabilities. It is assumed that the underlying Linux system is secure and that best practices are taken in the system's design such that a compromised ROS node cannot compromise the Linux system.

5.2.2 Attacker Model for ROS Deployment

The first adversarial model is an attacker model for the baseline ROS framework, which has no native security protection. There are three types of communication: node-to-node, node-master, or node-parameter server communication. Remember that all communication is unprotected in ROS, allowing an attacker to observe, inject, intercept, or modify communication between ROS nodes, between ROS nodes and the master server, or ROS nodes and parameter server. It is assumed that the attacker shares a network, and that they can

conduct any of the attacks described above. Specifically, an attacker can: (1) break confidentiality, as no communication is encrypted and (2) impersonate any participant, node, master, parameter server, as there is no authentication for the network message. An attacker can show up with any IP address, spoof connections to ports, send messages, and impersonate any node.

5.2.3 Attacker Model for SROS Deployment

The second adversarial model is an attacker model when the security extensions for ROS, SROS, are deployed. It is assumed that an attacker cannot break TLS and cannot easily gain access to the signing CA to fabricate additional certificates. Thus, a compromised node will have at most one compromised certificate. Each node is sandboxed with an AppArmor profile, meaning that AppArmor protects each non-compromised node. A node's AppArmor profile comes from the same source as a node and is therefore assumed to be as untrusted as the node. Thus, a compromised node cannot be trusted to be encapsulated by AppArmor, but every non-compromised node should be. It is assumed that the process protection takes the form of the `X.509` extensions and that the ROS system developer-only implements the minimum protections. The access control implementation requires detailed rules for each node to be deployed. Additionally, the whole system has to be redeployed when a rule is updated. It is assumed that most developers will only implement access control around critical nodes for ease of use.

5.3 ROS-Defender Design

The ROS-Defender system consists of three components: (1) ROSWatch, an anomaly detection system, (2) ROSDN, a prevention system, and (3) ROS-Policy-Language, a policy language. It allows specification of policy in ROS-applications semantics and mapping to network-level language. In this section, the design goals and an overview of ROS-Defenderare provided, then the main components are described in detail.

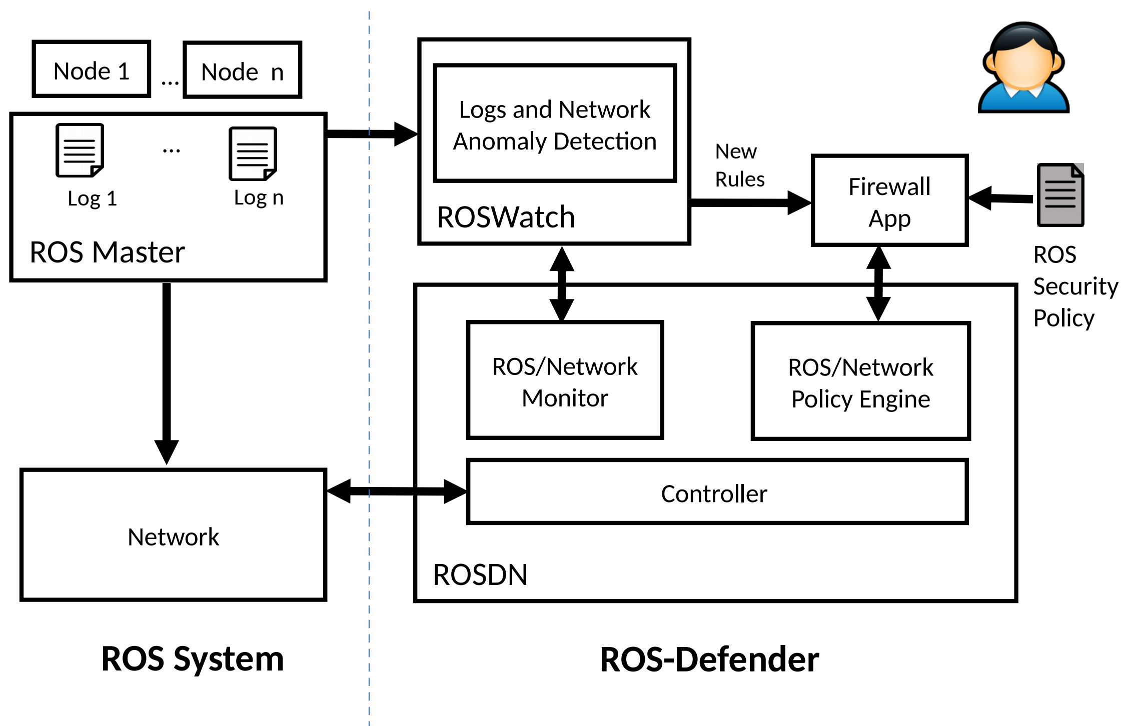


Figure 5.1: ROS-Defender Design

5.3.1 Design Goals and Overview

The decoupled architecture of ROS where the master maintains the logical view of the ROS graph and nodes communicate directly via the peer-to-peer overlay defined by this logical view provides benefits such as scalability, modularity, and flexibility. Unfortunately, it introduces fundamental security vulnerabilities as the entire system is abstracted onto a completely unprotected network. Protecting the network communication through the use of TLS prevents unauthorized nodes from becoming part of the network, while access control and application profiling – techniques proposed by SROS – prevent unauthorized parties from maliciously manipulating the ROS graph. However, these techniques do not provide mechanisms to verify at runtime that the system is compliant with a ROS graph policy and that the network level policy is consistent with the high-level topics and services policy. In addition, current solutions for ROS security do not protect compromised nodes or denial of service, do

not support reactive policies that are updated dynamically at runtime, and can not enforce low granularity network policies.

ROS-Defender is built with several design goals in mind.

- Provide network-level granularity protection. The network communication is targeted as the granularity of the system needs to support policies on a per-port basis.
- Allow for instant updates to the policy rules and creation of dynamic rules to support policy changes without system reboot and learn new rules that can prevent future attacks dynamically.
- Detect a large class of attacks such as compromised nodes and denial of service.
- Accommodate policies that do not require a ROS user to understand network-level rules, but only ROS level rules. i.e., topics, services based.
- Function without a full definition of the system for communication. The developer should not have to write every possible connection, as a rule, just the core functionality. The system should be able to react to other connections as it runs.
- Support both ROS and SROS. A solution that does not require API changes and provides protection for both ROS, and ROS running with the security extensions provided by SROS, is needed.

One way to detect network-level inconsistencies is by monitoring the communication between nodes. Enforcing dynamic reaction to network policies can be done by leveraging the recent paradigm of SDN, which provides a means to control network flows at Layer 2 in a network dynamically. Software-Defined Networking (SDN) is a novel network architecture that allows users to program traffic processing. It is divided into three different layers: application, control, and data plane. The application layer handles interfacing with the outside world and communicating data detected by the remaining layers. The control layer is the core of SDN; it handles communicating requests from the application to data plane layers, keeping track of all network functions, and monitoring all communications. The data plane layer handles the physical transmission of packets from one device to another[46]. However, as the targeted policies are application-layer policies, a mapping between these and the corresponding network-layer graph will be needed.

Figure 5.1 shows the design of the system, ROS-Defender. An intrusion prevention approach is taken where:

1. a user-defined policy is enforced at the network level with the help of ROSDN, which replaces the normal ROS network communication with an SDN approach to filter which nodes can communicate with other nodes as well as external connections, and

2. attacks are detected by ROSWatch, a log, and network-based anomaly detection which not only detects attacks but also learns rules that can use used to update the policy and prevent future attacks automatically.

A policy engine ensures that policies expressed in ROS-level abstractions by a user familiar with ROS are translated into network-level policies. The approach does not require any API changes or software alterations on the developer's part and provides security benefits to both ROS and SROS. Note that both ROSDN and ROSWatch can be used independently, but together they provide a much more robust security suite with defense, monitoring, detection, and dynamic reaction.

Examples in which ROS-Defender can improve security for both ROS and SROS include allowing for instant updates to the policy rules, creation of dynamic rules, being able to track if a malicious node does make connections behind master's *back*, filter malicious nodes, and for unencrypted communication, detect abnormal behavior from given topics.

Figure 5.1 displays the firewall application. The user defines a set of firewall rules using familiar ROS terminology. The ROSDN controller can understand this terminology through the use of the proposed domain-specific language. ROSDN treats the firewall as a continually updating policy, through the use of a predefined interface. When ROSWatch detects an anomaly, it would be able to update the firewall rules to correct the anomaly, such as blocking the packets from a compromised node. Below, the main components of ROS-Defenderare described.

5.3.2 ROSDN

One component of this approach is to replace the network switch with SDN in order to access all communication. The ROS middleware is centered around using existing IP com-

munications, adding security through the use of SDN is a natural fit. Using SDN provides fine-grained control over the network communications is granted, functioning as the backbone for ROS. Additionally, it allows for encapsulating the standard ROS communication paradigm, adding a communication white-list and monitoring.

The SDN component is referred to as ROSDN, as it has more functionality than an off-the-shelf SDN controller. Specifically, it includes two components a ROS/Network monitor that provides a ROS abstraction functionality for applications that want to operate on ROS abstractions (i.e., topics and services) and a policy engine that implements ROS-Policy-Language (see Figure 5.1). The ROS/Network Monitor maintains an internal state representation of the ROS system, tracking which open ports belong to which nodes and which open ports belong to which topics and services.

The ROS Policy Engine translates the domain-specific rules language into SDN understandable context, referred to as network-level policies. It leverages *textX* to define the language and the watchdog library to check the file for changes. OpenFlow protocol was used for SDN development. OpenFlow allows for building distinct flow tables, allowing the switch to handle packet flows with only communication to the controller required. These communications to the flow controller are only completed when there is a new stream; otherwise, the switch handles the processing.

ROSDN has two main functionalities: first extract the current state of ROS and then apply filtering policies according to the network-level policies that reflect the higher-level, user-specified, ROS policies. The state of the current system is extracted by locating and communicating with the ROS master. To ensure the accuracy of the ROS master, ROSDN also passively scans the open ports. Once ROSDN has extracted the information from the ROS master, it begins to filter new connections correctly. After these connections are filtered, all ROS communication will be correctly imported into a flow table. This blocks nodes from improper behavior and keeps track of the packet information in the system.

ROSDN supports two methods of flow filtering to create the flow table. The first method is to monitor the destination port of new connections. The destination port is then compared against the representation of the ROS system, as created by the ROS master node. In

using this measure, ROSDN is able to quickly and accurately tag new ROS connections and provide packet-level filtering. If the destination port cannot be identified, the controller first confirms that the representation of the ROS system is up to date. If the destination port still cannot be found, the controller defaults to a fall-back white-list and denies the connection if not found.

The second method is tied to the unique header in TCPROS[1]. Thanks to this unique header, the switch can easily identify all ROS communications, allowing the user to write ROS specific rules for metrics and targeting.

Using SDN with ROS provides many benefits for end-users beyond the security benefits outlined above. Openflow allows the user to collect several metrics about the flow of data through the network. These metrics grant a fantastic insight into the behavior of the robot. Additionally, many controllers support metadata tags of various flows, allowing tagging communication of various nodes by type. This, in turn, can be used to differentiate between topics and services and clearly mark which nodes are subscribed to which services.

5.3.3 ROSWatch

The second component of ROS-Defender is ROSWatch, a monitoring tool for detecting anomalies in network traffic between ROS nodes. ROSWatch takes as input network traffic and logs describing the state of the ROS system, and it identifies suspicious behavior and possible threats, attacks, and technical problems with minimal impact on users.

ROSWatch employs a pattern matching model for detecting network attack flows and logs using identifiers such as nodes and topics. ROSWatch's rules are classified into priority classes, based on a global notion of the potential impact of alerts that match each rule. The flow-level rules were constructed from the following features of flow records: average traffic volume, average publishing rate, average dropping rate, average/stddev/max age of messages, average/stddev/max duration of periods. These features provide the contextual modeling of the ROS data profiler used by ROSwatch. Rules are constructed in the following ways:

[1]This header is also detectable in UDPROS; however, that has not yet been fully implemented

- For numerical features such as the average traffic volume, the goal is to finely threshold them so that a rule specifying an exact number of flows can be properly captured. This rule takes the form "$feature > threshold \Rightarrow alert$".
- For categorical features like the node and topic name. This rule takes the form "if $node_i \notin NODE \Rightarrow alert$" where $NODE$ contains a set of nodes.

In addition to flow-level rules, log-level rules were constructed from the logs generated in each node, describing its behavior. This rule takes the form $if shutdown(node_i) \Rightarrow alert$. It means if the attacker shut down a node during the processing, then ROSwatch launches an alert.

5.3.4 ROS-Policy-Language

To grant ROS developers easy access to the SDN system's underlying components, a policy language that functions both at the application layer and the transport layer was devised. This policy language allows users to define rules that are source destination-based. A user can specify bandwidth requirements that leverage open-flow meter tables, check for encryption through the use of entropy estimation, and subsequently define the action taken when the rule is met. By default, the following are implemented: *allow*, *drop*, *log*, and *copy*. Source and destination use a custom-defined port object, either a transport layer port or a ROS node or topic name. These names will be translated invisibly behind the scenes. The bandwidth limitations create metered tables in OpenFlow, and they allow for priority fall-down so that the user can define several rules for bandwidth usage, (i.e., allow the first 100mb/s of bandwidth unimpeded and then drop everything above it). Additionally, a highly efficient version of the Entropy Estimation algorithm used by Dorfinger et al. [20] is adapted. Actions can take parameters, such as a destination port for *copy* and *log*.

Typically, the domain-specific code is designed to work with either one or two files. This allows users to use a single combined rules file or an application layer rules file with a separate transportation layer rules file. Additionally, an interface was created to allow external applications to take the roles of these files. This allows developers greater flexibility as they may use the domain language to speak to and control the robotic SDN for arbitrary applica-

tions.

5.3.5 Implementation

ROSDN is implemented using the OpenFlow v1.3 [26] implementation of Ryu [89] using the Zodiac WX [178] hardware. Building on top of the OverFlow protocol, Ryu was used for development.

Several features were added that made it easier to bring the system into a production system. The first of these is the ability to create templates for later import into the system. These templates allow developers to set up and test the SDN system once and become confident that it will work in the same way across multiple systems. Given that the SDN software replaces the ROS communication platform, it requires no alterations to the ROS codebase or to the security enhancements that may already be in place, such as SROS. The system was implemented to support importing the textual output from the ROS graph cleanly; thus, a user can easily transition from a development version to a production version by exporting the development ROS graph for the SDN system.

The domain language parser is implemented in textX as it allows for easy implementation of new policy languages. Additionally, implementation with the ability for users to define their own custom action and write their own python function for it was added.

5.4 Experimental Results

In this section, ROS-Defender is experimentally evaluated to demonstrate its functionality. The strength of the SDN platform and its support is established.

5.4.1 Experimental Setup

This system consists of a Turtlebot3 robot, a Northbound Zodiac Wx switch, and a laptop acting as both the SDN controller and ROS master/decision-maker. The data used was a SLAM path traversal of an already mapped area. The same area was mapped in Gazebo and performed the same path traversal. This allows for easy comparison between the virtual and physical robotic systems. The PC acts as both ROS master and SDN controller, with both sandboxed from each other through the use of *cgroups*.

The system is a purely virtual version of the Turtlebot3 system. It leverages the Gazebo package (a well-known robotics simulator) to emulate the sensor input for various robotic systems and uses *Mininet* for the networking component. This experiment was selected to compare the virtual system's performance with that of the real-world system and provide a broad base of robots for comparison. The SLAM algorithms were used for all robot types except the industrial robot, which was instead given a simple pick and place task. The PC is emulating the robot as a third *cgroup* sandbox. Different emulated robots of various complexities were analyzed in order to demonstrate the viability of ROS-Defender across several robot types:

- Turtlebot - Standard simulated robot for ROS platform[203]
- Husky - An outdoor rugged ground vehicle [189]
- AR Drone - Simulated Autopilot for the AR Drone platform [57]
- ABB industrial robot - Simulated pick and place industrial robot with ROS and Robot Studio [204]
- Udacity - Simulator for a self-driving car [207]

5.4.2 ROSDN Evaluation

ROSDN used .9% of the processor capacity and 50 MB of memory. For the robot system, the ROS/Networking monitoring thread takes 33 seconds to perform the first scan of the system, 1.28 seconds to check after a port is used that is not in the model, and an additional 15.2 seconds after it has returned the approval to update to the new model fully. If there is no need to update the model, it takes .005 seconds, on average, to add a new flow. On average, there are two flows added per topic connection (between the publisher and the subscriber) and one flow added per node (between the node and master). It used a maximum bandwidth of 2.18 MB/s during the first scan and an average bandwidth of 57 kB/s for updating nodes. The robot system design placed the OpenFlow controller connection on a separate connection from the data, which means that the OpenFlow overhead is not directly affecting the ROS messages.

The test systems were run in four separate states, without SROS or ROSDN, with SROS

Experiment Description	Avg. Latency	Max BW used	Time
RTurtlebot, No SROS or ROSDN	.0164s	190kB/s	46.7s
RTurtlebot, ROSDN	.024s	190kB/s	49.5s
RTurtlebot, SROS	NA	220kB/s	55.108s
RTurtlebot, SROS and ROSDN	NA	220kB/s	68.712s
VTurtlebot, No SROS or ROSDN	0.00426s	230 kB/s	47.1s
VTurtlebot, ROSDN	0.00568s	230 kB/s	57.429s
VTurtlebot, SROS	NA	290 kB/s	61.2s
VTurtlebot, SROS and ROSDN	NA	290 kB/s	75.432

Table 5.1: ROS Turtlebot experimental results, measuring the overhead of ROSDN and SROS and comparing the virtual system with the real. Note: SROS does not provide any way to measure the message latency so those values were omitted. RTurtlebot: Real Turtlebot. VTurtlebot: Virtual Turtlebot.

and no ROSDN, with ROSDN and no SROS, and with both ROSDN and SROS. For each run, each sensor's average latency was examined (i.e., the time from which something happened in the real world to when the robot was made aware), the maximum size a topic reached, and the time between system start and completion of the simulation. All of the results are summarized in Table 5.1.

ROSDN adds an overhead of 46% to the latency of sensor data for the physical experiment and 33% for the virtual experiment. Unfortunately, there was no ability to measure the additional latency imposed from SROS, as the tools used to measure it were not supported by SROS. The higher latency on the physical experiment is expected as the data is sent over a wireless network rather than through a simulator. The higher overhead reflects this as well. The message overhead was 15.74% from the use of SROS on the physical system and 26.08% on the virtual. If developers include a bandwidth overhead of 30% for the rule's limits of ROSDN, it will function equally well on both a ROS and SROS system.

ROSDN adds a startup overhead of 5.87% to the physical system and 23.93% to the virtual system, while SROS adds an overhead of 18% for physical and 29.94% for virtual. When both SROS and ROSDN were enabled, a final overhead of 47.13% (physical system) and 60.15% (virtual system) were added to the run-time. The higher overhead of the virtual system for ROSDN is due to the loss of a dedicated SDN system (the Zodiac WX), meaning that the flow processing had to be done on the laptop.

For the second set of purely virtual experiments, analysis of an already initialized robot was chosen to measure the additional overhead of ROSDN during normal operation. These results can be found in Table 5.2. The system's overall overhead for completing basic tasks was much smaller than the previously measured message overhead would suggest. As demonstrated in Table 5.2, the overhead of the time it took to complete a task was much smaller once ROSDN reached a steady state. This is likely due to the robot's run time being constrained more by the motor speed than the network processing speed. These results are comparable to those of other security research for ROS[116].

Experiment Description	**Time**	**Overhead**
Simulated Turtlebot, No ROSDN	42.101	0.0%
Simulated Turtlebot, ROSDN	46.5	10.45%
Simulated Husky, No ROSDN	33.59	0.0%
Simulated Husky, ROSDN	34.6025	3.014%
Simulated AR Drone, No ROSDN	23.68	0.0%
Simulated AR Drone, ROSDN	25.5	7.69%
Simulated ABB industrial robot, No ROSDN	17.62	0.0%
Simulated ABB industrial robot, ROSDN	19.14	8.62%
Simulated Udacity self driving car, No ROSDN	170.5	0.0%
Simulated Udacity self driving car, ROSDN	173.25	1.61%

Table 5.2: ROS Virtual Result comparisons

5.5 Conclusion and future work

ROS-Defender was developed as a comprehensive security architecture for ROS based robotic systems. ROS-Defender does not require changes to the existing ROS code since it is using available ROS application programming interfaces and SDN level mechanisms to monitor and execute access control actions.

This is a novel addition to the standard ROS system through the use of SDN techniques. This SDN modeled approach provides ROS systems with an added layer of security to prevent malicious attacks. The potential improvements to security and user experience such a system would provide were examined, and an example implementation was demonstrated. The experiments indicate that it is clear that ROS-Defender could supplement many security

flaws in ROS.

5.6 Key Contributions to ROS-Immunity

ROS-Defender led to unique insights into the challenges in securing ROS. During active testing, it was discovered that the start-up overhead heavily interfered with normal operations. Additionally, the requirement of a specialized SDN-enabled networking component proved to be a barrier-to-entry for normal ROS users, as it required restructuring of robot hardware. While ROS-Defender provided vastly improved security, the implementation costs were too high for the average ROS user.

However, ROS-Defender made it clear that not only is this type of security needed for ROS but that any such tool must be approachable for the average user. As discussed in the previous chapter, it was clear that ROS tools need to speak the same language as ROS to ensure that users could readily adapt to the tool. Further, any security systems need to alert a user (or another system) with the ability to remediate the attack rapidly and preferably before any harm is done. Lastly, the differentiation between ROSWatch, ROSDN, and the ROS-Policy-Language proved vital in guiding the system design. It was clear that parallel modular components were far more efficient than a single, monolithic approach.

In future chapters, the insights gained through ROS-Defender are further explored and improved, attempting to maintain the same security while rendering it more usable for ROS users. These improvements led to the development of the completed firewall component, as discussed in the previous chapter.

Chapter 6

ROS-FM

> *"There's no silver bullet solution with cybersecurity, a layered defense is the only viable defense."*
>
> James Scott

6.1 Introduction

In this chapter, the defensive firewall introduced in chapter 5 is extended to remove the dependency on external SDN components instead of taking advantage of novel Linux functionality by utilizing eBPF and XDP. The processing capabilities of XDP and eBPF are utilized to develop *ROS-FM*, a monitoring framework that provides a modular, scalable, and secure monitoring software for ROS that outperforms current solutions. The core functionality of *ROS-FM* is similar to ROS-Defender; however, the performance is vastly improved, and several structural changes were completed to address the remaining shortcomings of ROS-Defender.

rosfm's primary contribution includes the utilization of eBPF and XDP to process the ROS middleware efficiently. In recent years, there have been many advancements in process-monitoring and analysis for Linux systems in the form of the extended Berkeley Packet Filter (eBPF) and the eXpress Data Path (XDP). These new technologies allow for the execution of limited software in kernel space at lower points in the data travel path, allowing for efficient

and scalable processing.

The primary goal of *ROS-FM* was to build an extensible, efficient firewall with the ability to support numerous ROS systems. *ROS-FM* builds on existing network layer security for ROS-Defender but demonstrates the performance improvement available with the use of eBPF. It carries with it a domain-specific language for filtering rules, a more advanced modular framework that can easily hook into the ROS ecosystem with greater user functionality, and upgraded support for all current ROS and SROS versions[1]. This chapter demonstrates the power of eBPF/XDP in replacing the existing ROS command line tools with a monitoring system that can maintain metrics on every single topic and service within a ROS system while adding minimal overhead for the embedded systems. Further, the flexibility of *ROS-FM* is demonstrated by building two separate modules that target common use cases with ROS systems, security, and monitoring.

At present, there are many options available for monitoring multiple ROS systems, with the most common being AWS metrics [134] and Overseer [124]. While both of these options do support ROS monitoring capabilities, they both have performance comparable to the native command-line ROS tools. *ROS-FM* is shown to perform better than the native monitoring tools, which can be scaled to similarly large numbers of robots.

6.2 ROS monitoring tool

6.2.1 Design Goals and Overview

In this section, the implementation of *ROS-FM* is described. The tool was built to provide monitoring at its core and an extensible interface for other functionality. Both a security module and a data visualization module were implemented. The data visualization module is designed to export all of the monitoring and security information into Netflix's Vector[148] visualization for easy use.

The core monitoring module is built to monitor the entire ROS system, leveraging the performance available with eBPF. The security module is built on XDP and provides rule-based filtering for ROS systems and protection from both network-level and application-specific

[1] Included versions include ROS1, ROS2, SROS, and Secure ROS. This support excludes ROS-M.

attacks against ROS. In recent years, there have been many advancements in the field of process monitoring and analysis for Linux systems in the form of the extended Berkeley Packet Filter (eBPF) and the eXpress Data Path (XDP). These new technologies allow for the execution of limited software in kernel space at lower points in the data travel path, allowing for efficient and scalable processing. The Berkeley Packet Filter (BPF) [3] is a high-performance instruction set for a bytecode virtual machine inside the Linux kernel. It was created to be a fast programmatic network filtering tool in kernel space. BPF was extended in 2014 to what is now known as extended BPF (eBPF). eBPF [144] is an extension to the classic BPF framework with hooks to monitor processes beyond network applications. It can be used for generic event processing in the kernel, profiling programs, and libraries. This allows developers additional processing capabilities for more complex applications. One of the main advantages of eBPF is that it allows a user-space application to load code in the kernel at run-time. Thus, the new code is executed without recompiling the kernel or installing any optional kernel modules. The eBPF interpreter offers a limited selection of the standard C functionality, allowing the kernel to verify each section of the eBPF code formally. This limited selection does not allow loops and formally checks all memory accesses to ensure that the user-space code does not interfere with normal kernel functioning or normal applications' functioning. Several kernel hooks are available for the network stack, such as the traffic classifier `tc` [183], or the eXpress Data Path (XDP) [136, 106], a low-level hook component executed before the network layer, used for DDoS mitigation [129]. EXpress Data Path (XDP) [106] is a new programmable layer in the Linux kernel network stack, providing a run-time programmable fast packet processing interface inside the kernel and allowing users access to the lower layers of the networking stack to execute code to be run on packets. This allows XDP applications to reach performances far beyond traditional networking hardware.

The XDP system offers several compelling advantages over other kernel bypass solutions [106]. Specifically:

- XDP requires no changes to network configuration and management tools and retains the kernel security boundary.

- Ease in adding XDP execution hooks.
- Acceleration of critical performance paths when selecting the kernel network stack features such as the routing table and TCP stack.
- Transparency to applications running on the host, enabling the deployment of security rules such as DoS attacks.
- It can be dynamically re-programmed, i.e. the features can be added/removed on the fly without interruption of network traffic.
- Less overhead with lower CPU usage and power saving implications.
- It allows the user to custom and process packets that scale linearly with CPU cores.

6.2.2 Architecture Overview

ROS-FM is built as a central monitoring core and a collection of modules. As shown in Figure 6.1, the system works by integrating the ROS core communication code with the kernel. The eBPF code operates in kernel space to filter all ROS middleware communications. The **core** acts as a translation tool between eBPF and the ROS framework, providing users with multiple interfaces to manipulate ROS system data and traffic. To interact with and communicate with the core, a simple rule-based Domain-Specific Language is defined, discussed in Section 6.2.3, as well as data hooks. Two example modules were built to demonstrate the power of this construction, a **monitoring module** and a **security module**. A **visualization module** is utilized to export data for visualization in PCP. Other modules are possible, based on user input.

6.2.3 Domain-Specific Language

A domain-specific language that provides access to the underlying eBPF implementation is defined and implemented to facilitate user interactions with the system. This DSL is leveraged in both the monitoring and security modules. The language defines a set of simple rules that are automatically compiled into eBPF/XDP C by the core. This DSL is defined using the same primitives as ROS to allow for ROS users' ease of application.

Each rule in the *ROS-FM* system is written as a single line using the grammar outlined in Listing 6.1. The beginning of each line specifies the installed module to which the user wishes to apply the rule. As a user installs additional modules, new rules can easily be designed by simply extending the grammar specifying the new module. Once specified, an expression is constructed using ROS native syntax to aid the bridge between the systems. The user can chain any number of expressions together to meet a required specificity. As an example, if a user wishes to constrain access to the 'map' topic, to a set group of nodes(A, B, C), then the rules would be:

```
Filter: TOPIC == "map" && NODE == "A";
Filter: TOPIC == "map" && NODE == "B";
Filter: TOPIC == "map" && NODE == "C";
```

The majority of the operands are straightforward, regarding topics, nodes, and messages in the same way as the ROS framework. However, the LeftOperand 'CUSTOM' is used for module-defined behavior.

Listing 6.1: Domain-Specific Language Grammar

```
1  Rule:
2      (Module ':' Expression ';')+
3  ;
4
5  Module:
6      'Filter' | 'Monitor' | 'Display'
7  ;
8
9  Expression:
10     SimpleExpression '&&' Expression
11     | SimpleExpression
12 ;
13
```

```
14  SimpleExpression:
15      LeftOperand op RightOperand
16  ;
17
18  LeftOperand:
19      'TOPIC'
20      | 'NODE'
21      | 'MESSAGE'
22      | 'FIELD(' message_field=STRING ')'
23      | '{CUSTOM}'
24  ;
25
26  op:
27      Equal
28      | NotEqual
29      | Contains
30      | ContainedIn
31      | Matches
32  ;
33
34  Equal: '==';
35  NotEqual: '!=';
36  Contains: 'CONTAINS';
37  ContainedIn: 'IN';
38  Matches: 'MATCHES';
39
40  RightOperand:
41      STRING
42      | '[' (STRING)+ ']'
```

```
43 ;
```

6.2.4 eBPF/XDP for ROS

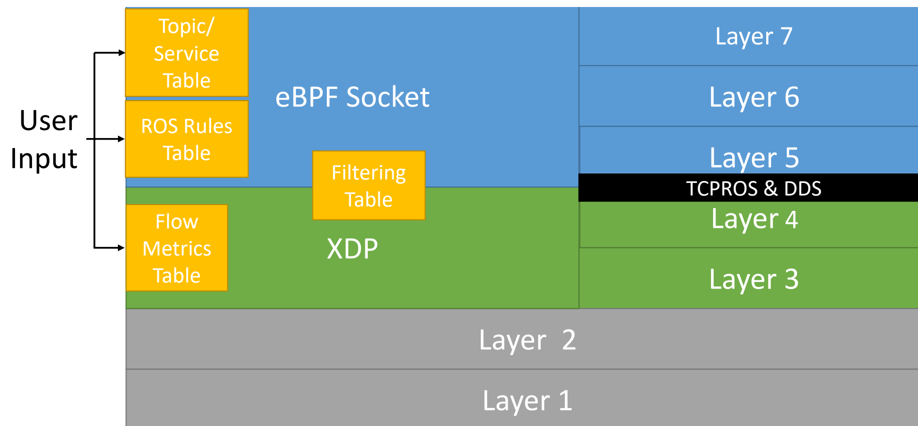


Figure 6.1: *ROS-FM*: Visual representation of monitoring system

The monitoring module operates on two different layers of the network stack, taking advantage of the capabilities of XDP and eBPF. The majority of filtering is done at the XDP level, while decision making is instead passed up to the eBPF socket layer and the main processing node. The program takes advantage of eBPF hash tables to extract data from the kernel. The first hash table is a mapping of a source port and destination port, with their respective IP addresses, with a rule for XDP filtering (Filtering Table). XDP uses the second table to keep track of any metrics the user desires (Flow Metrics Table). With every new packet, the XDP program updates the relevant stream for these metrics.

Any new packets that are not in the Flow Metrics Table are passed up to the socket layer with a meta-data note. The socket layer also maintains two tables of its own. It maintains a translation table for all of the topics and services(Topic/Service Table) as well as a lookup table for all of the rules for the security module(ROS Rules Table). The socket layer has two core functions: it analyzes all the incoming packets that the XDP layer is unsure of to determine what the topic is, and it maintains a lookup table to track the location of every

ROS component. Additionally, if the security module has been enabled, it takes advantage of the socket layer code to passively monitor all known node ports to ensure that the XMLRPC exploits do not function. Essentially, the socket program uses its privileged location to ensure that nodes do not have the API called out of turn and that any API calls are from known sources. A visual representation of this system can be seen in Figure 6.1, stacked against a representation of the OSI model for comparison. The ROS Rules Table and the Flow Metrics Table are the interfaces for the security and visualization modules.

The monitoring module provides the following custom commands:

- Log: Stores a copy of all matched packets to disk with the location specified by a user as a RightOperand
- Count: Maintains a running count of all packets that match the filter
- Average: Maintains a running average of all bytes per second that match the filter.
- Max: Stores the current largest packet that matches the filter.
- Min: Stores the current smallest packet that matches the filter.

ROS2 is built on top of the Data Distribution Service (DDS), which still operates on the normal Layer 1-4 network stack. This means that the XDP layer is identical between ROS1 and ROS2, while the socket layer only has to parse different packet meta-data. Additionally, it includes implementing both a TCPROS/UDPROS filter and a DDS filter for the socket filter, as two separate eBPF programs. While both of these programs can be loaded simultaneously, the filters cannot trace packets across the *ros1-bridge* node, meaning that such traffic is lost. This was considered acceptable as the *ros1-bridge* appears to be rarely used, as do hybrid systems in general.

While eBPF places substantive limitations on both the size and complexity of programs, this system was designed so that higher-order processing was done purely in user-space, with the kernel space programs following the rules set in place in user-space. This allows for a more extensive development environment, though it does come with the limitation of a delay before new rules are fully implemented.

For the security module, both eBPF and XDP is leveraged to drop packets faster than user-space programs. The security nodule extends the monitoring capabilities at both the socket and XDP layers to add application-specific content. This is most notable in the socket layer where the security module, even with no other rules, enforces the ROS assumption for communication (a node must first communicate with a master node before it attempts to establish a connection with another node). This cuts off a wide variety of XML-RPC attacks against ROS, which tend to function by exploiting the API. This is a vital function ROS does not provide. As an additional feature, the user is provided the ability to restrict access to specific function calls so that an attacker cannot update command line parameters or call a service it is not supposed to under penalty of dropped packets.

The firewall is configured as a default-deny filter for all topics and nodes it has a rule for, and default-allow for those without a rule. The firewall provides the following custom commands as used in ROSDefender:

- Copy: Transmits a copy of all packets to another destination
- Limit: Allows the packets to pass up to the limit then drops the rest

The functionality of XDP is utilized to allow for the enforcement of consistent origins for messages from nodes. Once a node has been identified on the network, any attempts to spoof that node from a different source address are detected and blocked. This functionality extends to anonymous nodes as there is no intention to restrict any ROS features.

6.2.5 Visualization Module

The visualization module is designed to give users a consistently updating sense of their robots' network traffic and state. It is designed to allow each robot to automatically export every metric that is available through the robot command-line interfaces, with low overhead. This module exports data in the PCP format.

Netflix's Vector's ability to pull and plot data from a large number of sources is leveraged to export chosen metrics from multiple robots to a convenient location. This allows users to have an intuitive sense for their robots and quickly respond to any issues or attacks.

6.2.6 Implementation

ROS-FM is implemented in Go using gobpf to access the eBPF and XDP bindings. Go was chosen due to its higher performance [185]. Each of the system's components is encapsulated in their threads, with shared data buffers to share information between them. The tool leverages many efficient parallelization mechanisms in Go, including the channels and the dynamic rewrite semaphores. The core monitoring program provides an API for the other subsystems to provide a consistent interface for any modules.

The security module is implemented as an extension to the table translation interface between topics and services. Given a defined rule file, which is structured to be similar to a ROS launch file, a user can specify which nodes are allowed to communicate on which topics, call which services, and how much data nodes are allowed to use to protect against DDOS and other broadscale network attacks. The rules file for the security tool does not have to be static and can be updated regularly to take advantage of existing research in machine learning and anomaly detection for ROS, such as with ROSWatch (Chapter 5).

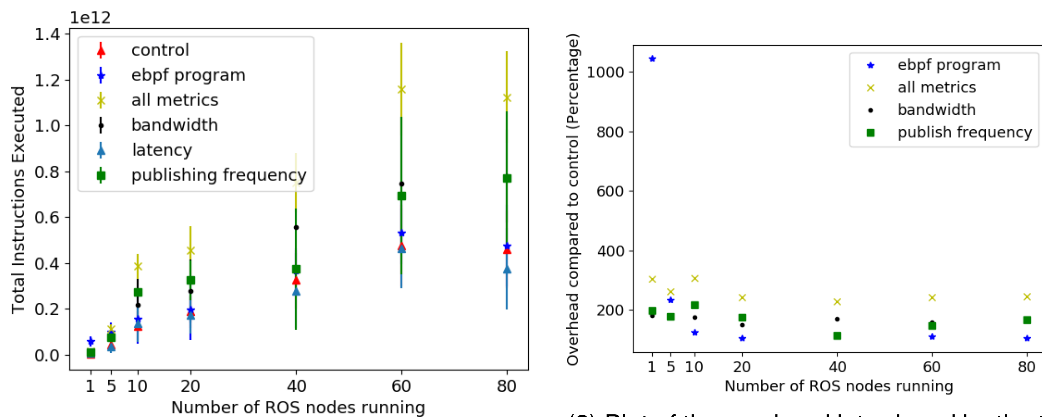
The visualization module is implemented as a PDMA plug-in using the Go PDMA library. It exports to the *mmv* module for PDMA, which is designed for custom user plug-ins. The *mmv* module uses shared memory to monitor processes on the system. From there, any data visualization tool that can process PCP can render the data. For the visualization in Vector, templates were built to render the most common ROS metrics: number of topics, packets per topic, bandwidth per topic, and number of nodes and subscribers on a topic over time.

6.3 Experiments

The following research questions were defined:

- Q1. What is the overhead of using *ROS-FM* vs. the state-of-the-art ROS tools?
- Q2. What is the break-even point between the ROS tools and *ROS-FM*?
- Q3. How effective is *ROS-FM* at preventing common ROS network attacks?

These questions were addressed by performing the experiments described in this section.



(1) Overhead comparison of ROSCLI tools with ebpf program

(2) Plot of the overhead introduced by the the CLI tools vs the ebf system

Figure 6.2: Analysis of ROS1 tools vs *ROS-FM*

6.3.1 Experimental Setup

The monitoring system was implemented in Go, using the gobpf library to load the eBPF code. The eBPF code was developed in C and built at run time within the code. All experiments were carried out on a 3.2GHz Intel Xeon E5 virtual machine, 8 GB RAM, running Ubuntu 18.04, running kernel 5.0.0. ROS1 Melodic and ROS2 Crystal were used, using the release versions of each. All tests were carried out with the security and display modules to create a 'worst case' measure of overhead.

The following experiments were conducted:

- Compare the overhead induced by the rostopic CLI tools with the overhead induced by the eBPF tools. The overhead is tested in terms of processing cycles introduced on a ROS system with 1, 5, 10, 20, 40, 60, and 80 nodes communicating on one topic.
- Repeat the overhead measures with more topics, invoking the CLI tools on every single topic.
- Compare the accuracy of the eBPF performance tools against the Amazon ROS performance tools.

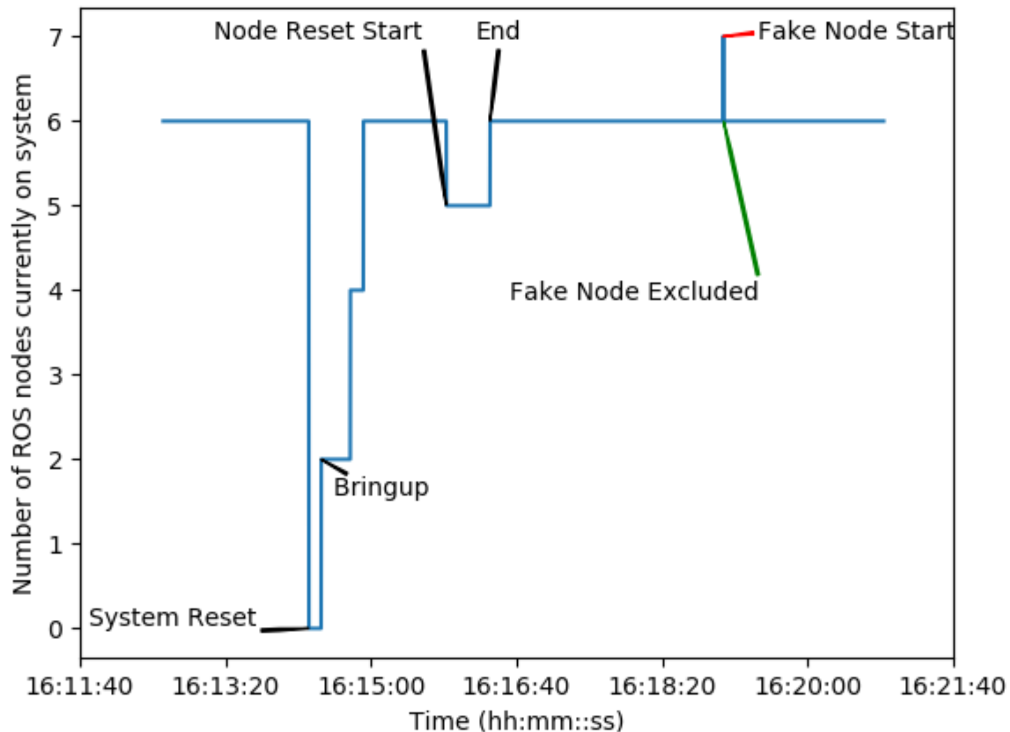


Figure 6.3: *ROS-FM*: Node tracking example

- Evaluate the overhead on a ROS2 system.
- Using industry-standard security tools ROSPenTo and ROS2-SecTest to determine how effective the security plugin is.

6.3.2 Overhead analysis

In analyzing ROS1 overhead, the system has a processor overhead between 660% for a ROS system with a single publish/subscribe node pair and 1.6% for a system with 80 publish-subscribe nodes. It is important to note that these are the 'worst' and 'best' case scenarios, respectively. In the average case (5-8 nodes), the system performs comparably to all ROS metric tools while simultaneously providing additional security. However, if the user is only concerned with a single ROS metric, this tool provides the benefit of added security but has

a moderately higher performance overhead. In cases with ten nodes or more, this tool far outperforms all others with an overhead of 23%, compared to 73% for a single topic or 205% for all metrics.

The high-performance overhead, when compared to the single node system, is due to the overhead of the userspace program and the eBPF compiler. The singular ROSTopic metric's performance overhead is found to have an overhead between 34% and 52%. Once all of the ROStopic metrics provided by the eBPF program are included, the performance overhead of the CLI tools instead ranges from 105% and 140% depending on the number of nodes. Once there are more than five node pairs on the system, the eBPF program outperforms the equivalent ROS tools needed to provide the same metrics. Additionally, the eBPF program exceeds the monitoring tools' performance completely after ten node pairs have been added to the system. On the far end of the spectrum, the eBPF monitor outperforms the standard ROS tools by a factor of 25. These overheads are demonstrated in Figures 6.21 and 6.22, with Figure 6.21 demonstrating how many instructions were used over 120 seconds of run-time and Figure 6.22 demonstrating the overhead of this tool and the command line tools as compared to the control 120 second run.

In ROS2, the overhead is between 515% and 15% for the eBPF program and 80% to 110% for the ROS2 CLI tools. The current ROS2 master and node paradigm had a higher startup cost, but that was offset by the overhead of parsing the DDS packets. Findings in ROS2 were analyzed using the eProsima DDS provider.

6.3.3 Security Evaluation

In order to evaluate the security provided by this solution, it was compared to two existing security tools for ROS1 and ROS2, ROSPenTo[164] and Amazon's ROS2 Security test node[135]. These two systems were the current best standard test tools available as of 2020, though, like all penetration tools, they are only a starting point for security comparison. The two tools were activated on a system secured with the XDP module and the results are summarized in the Tables 6.1 and 6.2. It was discovered that it was possible to completely negate any attacks that relied on out of order execution of exploits to the node API

Attack	Result
Add publisher	Negated
Replace publisher list	Negated
Remove publisher	Negated
Isolate Service	Negated
Unsubscribe Node from Parameter	Limited
Update parameter	Limited

Table 6.1: *ROS-FM*: Effectiveness of *ROS-FM*on ROSPenTO

CPU Exhaustion	Successful
Network Exhaustion	Negated
Disk Exhaustion	Successful
Memory Exhaustion	Successful

Table 6.2: *ROS-FM*: Effectiveness of *ROS-FM*on AWS ROS2 SEC Test Node

directly, such as those attacks that change the publisher or subscriber status. Additionally, it was relatively simple to protect services from isolation and calls by unauthorized nodes. While attacks between nodes and the parameter server were still possible, they were rendered more difficult as they could only be launched from the same source as the valid node, and they could only affect the parameters to which the node has access. The addition of PKI renders all attacks against the parameter server impossible. Unfortunately, the system is far less effective against the ROS2 SEC-Test attacks, as it is only possible to negate the network resource exhaustion effects and provide no protection against the other resource exhaustion effects at present. A future extension to the security module could take advantage of eBPF's socket options in that area to limit nodes' access to computing resources.

It was confirmed that the addition of Secure-ROS, SROS, and the secure DDS extension for eProsima do not heavily impact the eBPF monitoring tool's performance. However, the socket layer analysis is more limited in cases where encryption is used. Once the packets are encrypted, it is no longer possible to enforce limitations on the XMLRPC calls or any other rules that require knowledge of the packets.

6.3.4 Monitoring Evaluation

The appearance of the exported data for the number of topics active on a system during normal operation is demonstrated in Figure 6.3. The PCP plugin also exports the current number of packets sent on the topic, the current bandwidth used by a topic, and the average packet size per-topic. All of these topic-specific metrics are dynamically added and removed from the PDMA exporter at runtime.

6.3.5 Comparison and discussion

It was discovered that the eBPF tool exceeded the performance of the traditional ROS CLI tools once the system reached a sufficient level of complexity, and the initial overhead of loading eBPF is proportionally small. Additionally, it was possible to monitor both ROS1 and ROS2 with similar overhead, though ROS2 carries a higher overall overhead. This was all demonstrated while providing both security protection and metric visualization for users.

6.4 Conclusion and future work

In this chapter, *ROS-FM* was described as a system to monitor robotic systems built on top of ROS. This system provides better performance than the native ROS1 tools and still supports the same monitoring for ROS2. Furthermore, two modular extensions to the system were introduced, a security module and a data visualization module that leverages Netflix's Vector tool to render the results for a more straightforward user experience. The security module was evaluated against standard security tools for both ROS1 and ROS2 and show the improvements gained from using this system.

While *ROS-FM* can be added to any singular robotic system, there are some possible future research improvements. Firstly, testing the system with additional ROS2's DDS implementations to evaluate what other metrics are valuable to track is needed. Furthermore, expansion of the system to cover multiple separate robots would be useful to make it possible for them to share their hash tables and rules to track distributed robotic systems. Additionally, building other modules to address other needs in the robotics community with eBPF may be useful.

The main contributions of *ROS-FM* are:

- Monitoring: An eBPF monitoring framework for robotic systems with very low overhead.
- Security: A network layer security add-on to extend the core ROS security model.
- Usability: A PCP plugin to export and easily visualize robotic statistics.
- Reproducibility: The code and data are publicly available.

6.5 Key Contributions to ROS-Immunity

ROS-FM composes the firewall and monitoring components of *ROS-Immunity*. The framework of *ROS-FM* is utilized as the defensive firewall for *ROS-Immunity* , with an update for coordinating across multiple robots. Further, when coupled with the autoencoder implementation discussed in Chapter 8, the two components also act as the core anomaly detector. The finalized firewall filters traffic, checks for potential abnormalities, and secures robots from known vulnerabilities.

As stated, the firewall in *ROS-Immunity* was updated to include coordination across multiple robots, a feature not included in *ROS-FM*. This feature was crucial to address security issues in distributed robotic systems. *ROS-FM* was initially designed as a tool for a singular robot and did not include this functionality. However, the modular framework supported the extension of *ROS-FM* with minimal issues due to the ease of interfacing with ROS traffic using eBPF.

Chapter 7

DNS Privacy: A Use-Case for *ROS-FM*

> *"Arguing that you don't care about the right to privacy because you have nothing to hide is no different than saying you don't care about free speech because you have nothing to say."*
>
> Edward Snowden

In this chapter, the flexibility and power of *ROS-FM* are demonstrated with a use-case scenario. DNS privacy was chosen as a highly applicable use-case, as DNS queries of users and companies contain a plethora of sensitive information with very little privacy.

DNS is a fundamental protocol on the Internet, and as all traffic originates as a DNS request, users reasonably expect that these queries remain private between the stub resolver and the DNS nameserver (even though these queries are typically sent in cleartext). DNS privacy is a fundamental tenet of browser privacy as all traffic originates as a DNS request, and anything embarrassing or potentially compromising from a user is visible to the browser and vulnerable at the DNS layer. Attackers who successfully intercept DNS communication gain unique insights into user behavior. Despite research in areas of resolver privacy and encrypted traffic [139][105][80][58][42], most current solutions in protecting DNS traffic are at the provider level, outside the control of users. This allows providers to access

any information from a DNS request, potentially decreasing a user's privacy. RFC 7626 [42] recommends that one of the best options for privacy-conscious users is to use different DNS providers for different applications(e.g., Firefox, Skype), rendering them more difficult to track. The current structure of most user-based systems makes using multiple DNS servers difficult, prompting the need for a more efficient way to utilize this privacy method.

Currently, most DNS privacy applications are designed to protect against third party eavesdroppers, but there is not a similar system for users to control their privacy against pervasive monitoring at the resolver level. Privacy-conscious users who wish to avoid being fingerprinted by their DNS provider must manually change their system resolver, a very coarse solution. In this chapter, *ROS-FM* is leveraged to provide an alternative system to route user DNS queries to various providers. This addresses the current need for an effective method for privacy-conscious users to control their DNS traffic and increase privacy. This system should also be compatible with other DNS user privacy advancements, such as encrypting user data over HTTPS or TLS.

The goal is to provide users with more control over their privacy by allowing them to use application-specific DNS filtering or monitoring for any form of DNS that cannot be directly filtered. This demonstrates the capabilities of rosfm's user-communication plug-ins, allowing the filtering system to communicate information to the user trivially. This provides users with far greater control of their DNS queries.

7.1 Threat Model

There are four groups of potential threats to user privacy considered for the threat model applying *ROS-FM* to DNS: privacy from the ISP, the browser vendor, attackers on a shared network, attackers on a shared system. Each threat can be modeled as an actor with different capabilities and interests. This threat analysis assumes all four possible threats, although that is not the common case. It is also assumed that the user is not employing any other DNS privacy tool besides DoH and DoT. With this in mind, the current protections provided by DoT and DoH for each group are summarized:

1. The ISP: The ISP is the most significant concern to user privacy due to the amount of

control over a user's data. The ISP can control the stub resolver for the host, monitor any traffic that the host publishes, and even monitor the sites the user visits. Privacy from the ISP is the most complicated to address as it requires a high level of encryption and anonymization on all traffic entering and leaving the network. The ISP utilizing user information to compile a user profile is considered a significant threat to user privacy. This threat model assumes that the ISP can determine any DNS transaction to its stub resolver. However, DoH and DoT are assumed to provide protection if an alternative resolver is used than the resolver owned by the ISP. Due to the ISP's scope of capabilities, it is considered the most significant threat.

2. The browser vendor (e.g., Google, Mozilla, Apple, Microsoft): The browser vendor has access to all traffic conducted by the user in the browser. If left unsupervised, the browser poses a significant threat to user privacy. For this threat model, the privacy risks of the browser vendor are only considered concerning DNS privacy when users utilize DoH, as this is the sole instance when the browser itself has control over DNS queries. Here, the focus is placed on the browser vendor as there is already an existing ecosystem of DoH providers who are also browser vendors. DoH provides users increased security from the ISP while decreasing browser security by providing the browser vendor with complete access to user DNS transactions. Since DoH bypasses cleartext DNS and DoT, the browser vendor is in complete control and can gather information freely. Similar to the threats to user privacy discussed with the ISP, the browser vendor compiling a user profile is considered a threat. The threat model focuses on the security risks in the browser vendor caused by the solution provided by DoH.

3. Network Eavesdroppers: On shared networks, DNS requests may be accessed by other users due to traffic monitoring or flaws in encryption. Network eavesdroppers are perhaps the most common type of attacker, and therefore their capabilities are the best understood. Network eavesdroppers are included within the threat model but are well handled by DoT and DoH as both provide well-established solutions to these problems.

4. Server Eavesdroppers: Server eavesdropping is more advanced than network eavesdropping and requires the attacker to be on the same system as the user. This can occur in a cloud environment or through other shared servers. Server eavesdroppers have all of the same capabilities as network eavesdroppers; however, they can also perform new types of attacks relating to user searches and caches. Using only DoT does not address these issues because of shared network caches, but DoH or false DNS queries can mitigate it. Any solution must be conducted in a more privileged ring of escalation to be considered viable for these threats.

For this chapter, it is assumed that only the root user (any user with permission to run eBPF programs) can be trusted with all user's DNS queries. Any other users or permissions are assumed to be a potential threat. This chapter follows the model provided in Kim et al. [45] that determines the amount of information leaked in a given query. The goal is to minimize the number of bits of information that anyone DNS server receives, thus limiting opportunities to build user profiles.

7.1.1 Attack Descriptions

This chapter considers the following attacks:

- *De-anonymization*: De-anonymization attacks allow leakage of personal information that connects users to queries or creates profiles of users based on their queries. This attack is most frequently observed on the ISP or the browser vendor. Neither DoT nor DoH currently can address this type of attack. For this chapter, the goal is to ensure that a given resolver cannot construct an accurate profile of the user.

- *DNS Query Leakage*: DNS Query Leakage attacks occur when any part of the DNS query is leaked to any observer aside from the resolver. This can occur due to a lack of encryption, timing attacks, or other similar attacks. DoT provides encryption solutions but cannot address timing attacks, while DoH addresses encryption and timing attacks.

- *Man-in-the-Middle*: Man-in-the-Middle attacks occur if an attacker can intercept DNS communications. Both DoT and DoH use TLS to address this.

- *Malicious Resolver*: Malicious Resolver attacks are DNS resolvers that return incorrect IP addresses though the error response mechanism. Neither DoT nor DoH has a way to interact with this.

- *DNS spoofing*: DNS spoofing attacks also return incorrect IP addresses; however, they are returned by a server downstream from a DNS resolver. While DoT and DoH do not directly provide security for these attacks, the infrastructure required to enable DoT and DoH provides security. Standard encryption and certification methods address this attack.

7.2 Methodology

In this section, the various ways eBPF can be leveraged to aid privacy-conscious users in protecting their privacy are described.

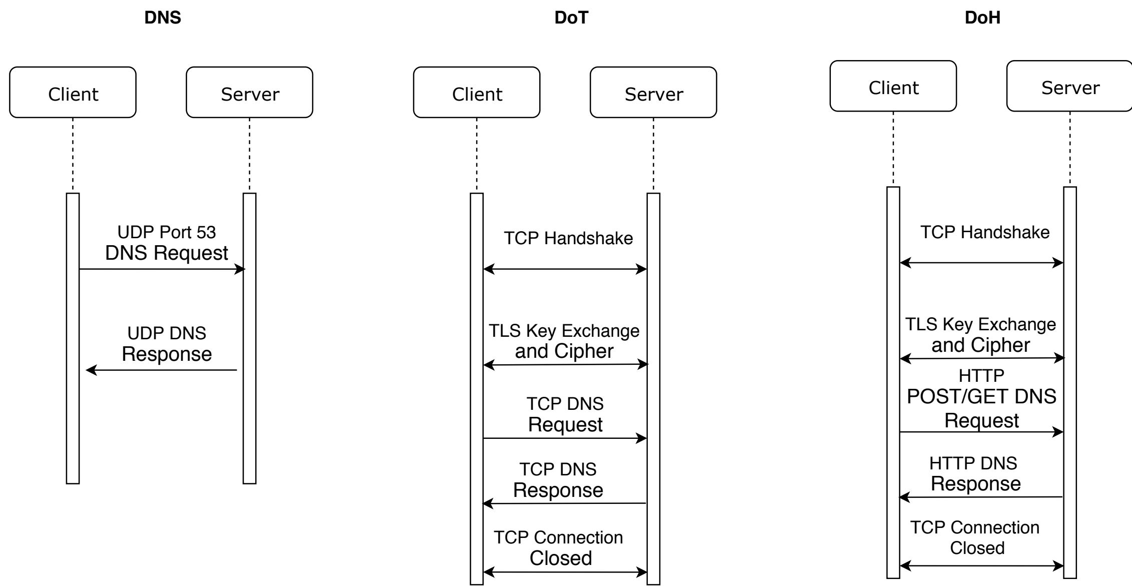
7.2.1 Design Goals and Overview

eBPF provides developers with far greater access and control over user-space programs, which can be leveraged to provide greater privacy protection of users' DNS requests. The potential benefits provided by eBPF for traditional DNS, as well as the newer DoT and DoH, are analyzed. The goal is to use the functionalities of eBPF to give the user more control over their DNS privacy.

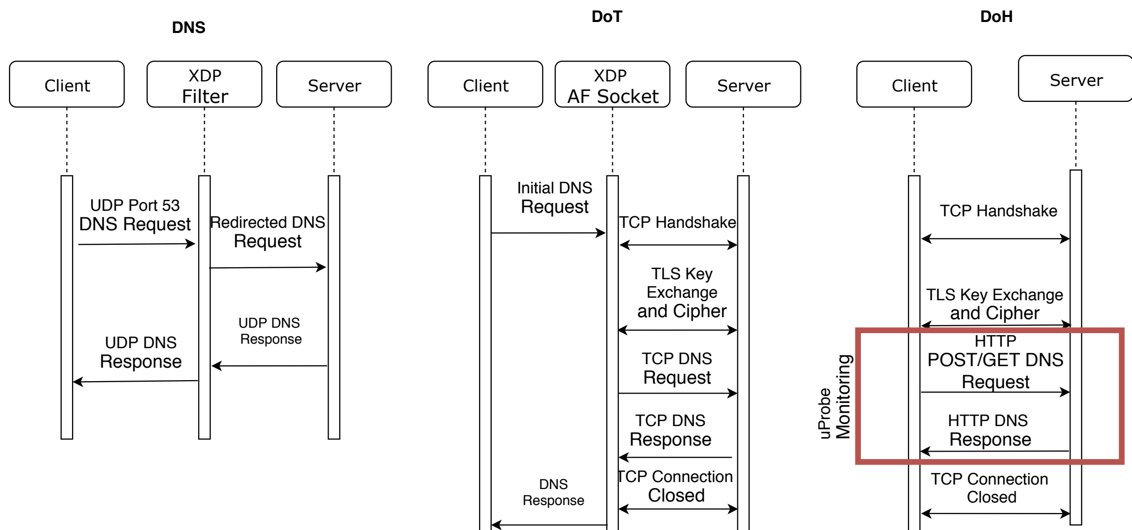
The core of this research is the use of eBPF to provide users with an application-specific DNS implementation by adding functionality to invisibly edit the requests so that each application can point to an arbitrary server. A method is demonstrated to capture every single DNS request and forward it to a group of other servers to detect potential prioritization or differences between responses.

7.2.2 Implementation

The DNS system was implemented in Python 3.6 on top of Ubuntu 18.04.3, with kernel 5.0.0.27 using python-bcc[145]. The system was implemented using a hash table of process names and destinations. The system uses XDP to edit the packets before transmission. The name of the application is hashed in eBPF, and the system's ability to determine which



(a) Current versions



(b) The eBPF implementation

Figure 7.1: Normal functionality of DNS, DoT, and DoH vs. eBPF implementation

program is sending a given packet is leveraged. Then, it is determined if the packet is a DNS request and apply the appropriate filtering depending on the hash table rules. By default, the system DNS server is used if there is no specific rule to be found. It is implemented

application-specific because it was the most structurally complicated and, therefore, the best demonstration of eBPF capabilities.

For DoT implementation, the eBPF AF socket, a kernel-level socket with instantaneous data communication between kernel and user-space, is used. The TCP packet is intercepted before it leaves the system and changes the DNS server's destination address. The negotiation is completed by acting as a trusted MiTM. In this way, the user-space program usually behaves but does not transmit outside of the system. Structurally, an application-specific DoT system was built in the same way the DNS system was built. However, there is more complexity added due to the TLS handshake and encryption requirements.

Additionally, eBPF uProbes were leveraged to hook into the DoH encryption, and decryption for monitoring sent requests. In the uProbes, it is possible to see the arguments between the encode and decode functions and their respective return values. While it is not possible to edit any of these, this makes it possible to reconstruct the message that is sent and display its contents to the user. It is also possible to construct and send fake DoH requests. It is thought that this is not valuable since the Browser vendor operates the DoH server and can quickly verify that the fake requests are essentially noise and discard them. This means that any non-privacy conscious actors can quickly verify the real lookups. Therefore, the focus is not placed on this functionality.

For all three versions of this tool (eBPF for DNS, eBPF for DoT, eBPF for DoH), the 1BD-improved and EFF's panopticlick algorithms are utilized [140]. *ROS-FM* uses the algorithms to determine how much information is leaked to a given DNS resolver to determine to which server to send the next packet of information. The number of bits exposed is calculated in real-time. A list of resolved domain names is saved to prioritize the sending of information of similar queries to the same DNS resolver, to reduce leaking information to multiple resolvers. In DNS and DoT, this list is leveraged to minimize the number of bits exposed. For example, if a user were querying the top Alexa 500 site, this eBPF system would choose the resolver for which visiting the site would reveal the least amount of information about the user. Subsequent revisits to the same site use the resolver that satisfied the first query. However, in DoH, as this behavior cannot be manipulated, the amount of bits exposed is

calculated and shared with the user. In order to allow users to interact with the system, a simple application is provided that identifies running applications and allows users to specify which DNS or DoT server to which they should connect. This application also monitors DoH and displays an estimation of leaked information.

7.2.3 eBPF for DNS

Traditionally, the best method for concealing DNS activity was to use the most popular DNS server and 'hide' in the noise of the massive amount of requests they received every day. However, with the recent advent of big data processing technology, this strategy is no longer valid. As such, the best strategy has become to shift queries to a wide variety of competing DNS servers, providing an incomplete or limited picture of the user's browser activity to any single server and making tracking the user more difficult. In order to spread traffic, the current solution is to use a DNS *sink hole* [16] or a similar router-level solution, as the system DNS is not designed to support arbitrary server changes. However, these systems require control over the router, and on complex networks using this method is not feasible and does not provide extra privacy.

With the advent of XDP, there is now an efficient, low-cost way for users to route DNS traffic to arbitrary servers at the system level. XDP is a component of eBPF that has been extended to be able to not only monitor but also change data. As eBPF is automatically aware of which process is sending a packet and can easily find the type and content of an ordinary packet, users can now filter DNS requests by applications. As shown in Figure 7.1, all requests and responses are filtered through the XDP filter before being sent to the server(or returning to the user). This masks activity by directing different requests to different servers. eBPF is so versatile that a user can filter the application, the time when it was sent, or even maintain a circular buffer of acceptable servers and send one request to each in turn.

Even with eBPF, users are not protected from Network Eavesdroppers nor ISP monitoring due to the lack of encryption on DNS. However, DoT and DoH are designed to address this issue.

7.2.4 eBPF for DoT

DoT is effectively the same as traditional DNS, but it is encrypted using TLS. Thus, it provides privacy from network eavesdroppers and ISPs concerning the content of browsing.

The eBPF functionality called an AF socket is leveraged, modeled off of the previous *ktls* work [206]. This creates a virtual socket to route all traffic through. In this way, the DoT request is captured, edited to direct it to the specified server, and then sent to the virtual socket for handshake and communication. Once the final reply is obtained, the socket is connected back to the normal application. From the application's perspective, it merely performs a DoT query and receives a result as it usually would, but underneath it, it is possible to change the DNS server it communicates to. For example, using eBPF in this way, an application may request a query from one server, but it is intercepted before the first TLS request is sent and instead sent to a server chosen by the user. This allows the user to send activity to an arbitrary DoT server much like they would a DNS server, as shown in Figure 7.1. This method makes it possible to do the same application-specific filtering as traditional DNS in a DoT environment.

7.2.5 eBPF for DoH

DoH gives users more privacy from ISPs at the cost of less privacy to browser vendors. Since DoH uses purely browser-internal functionalities (encrypted GET/POST), it makes DNS requests from browsers problematic to analyze from the outside. At the moment, there is no known way for a user to monitor their DoH requests and see precisely the data being submitted, except for through the browser's debug console. This means a user has no control over the DNS data being generated on their behalf.

While eBPF does not allow developers to directly interfere with DoH requests like with normal DNS requests, the monitoring and tracking capabilities provided by uProbes can be leveraged to monitor the creation of the DoH requests and the decoding of the responses. This allows the user to verify that the browsers are not transmitting extra data and alert users about how much a given DoH provider knows.

While it is not possible to provide application-specific DNS privacy with DoH systems,

sending fake requests to the server obfuscates the request of interest. Theoretically, as the browser still knows which request is the truth, it can still track the user from source to destination. Therefore, this is not a full solution for direct privacy to DoH users.

However, the key advancement of this approach is that it can be used to monitor functionality. The user can use this method to view the information the browser is providing about DNS lookups to monitor all information shared by the browser. This allows the user to know exactly what information is being transmitted by the browser, which can be used to determine when it is time to change DoH providers, or when it is time to disable DoH and take advantage of the application-specific eBPF for DNS or use DoT to preserve privacy. Additionally, it can guarantee that the browser is only sending information to the chosen DoH provider and not leaking information to other systems. This provides the user with direct control over their DoH usage and reduces privacy risks at both the ISP and browser levels.

7.3 Experiments & Results

The performance impact is analyzed by using eBPF to route all DNS traffic to different servers. The system is tested with application-specific and time-based routing. Three experiments evaluate the system from a privacy perspective and quantify the overhead introduced by the system.

7.3.1 Performance Experiment Setup

The overview of the experimental setup is outlined in Figure 7.2. To verify the application-specific DNS code was working and not detectable, the system was set up with four applications pointing to four different DNS servers. Applications used included Firefox, Curl, nslookup, and wget. The four DNS servers used were 8.8.8.8 (Google), 1.1.1.1 (Cloudflare), 9.9.9.9 (Quad9), and 158.64.1.29 (Fondation RESTENA). Both Wireshark and a virtual switch were used to record all DNS traffic to verify the packet filtering and simulate both types of adversaries. To verify that the filtering was not detectable by the applications, their debug output and warnings were recorded, and it was confirmed that there were no new errors emitted by the application. For the experiments, only IPV4 was used.

Additionally, the eBPF system was evaluated while it was monitoring DoH packets. While

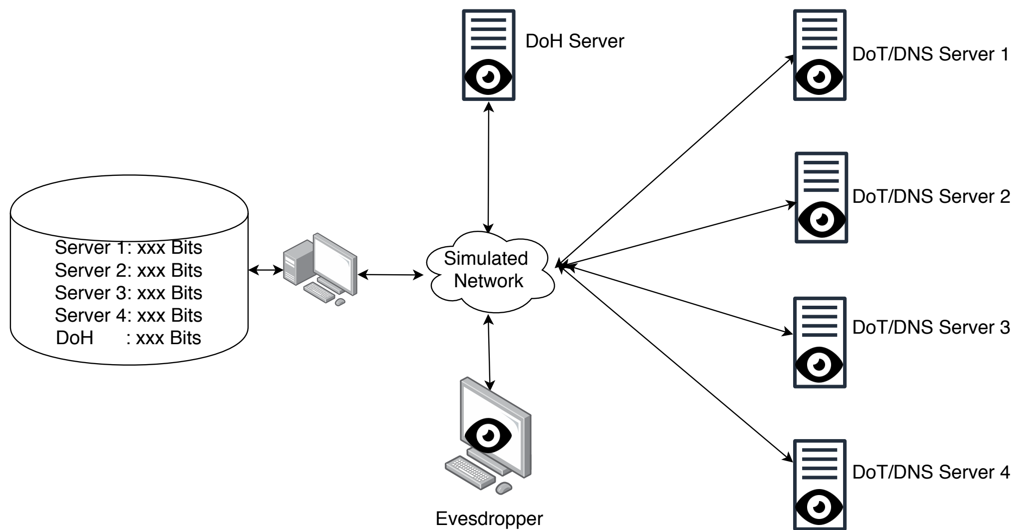


Figure 7.2: *ROS-FM* and eBPF: Experimental Network overview

it was not possible to entirely redirect the DoH packets with the current limitations of uProbes, the requests were monitored and returns verified. Returns were exported to the user, and an alert was raised if there was a deviation between the DoH return and system lookup. While such a system would not be used in practice, it was helpful to confirm that the DoH providers were not trying to shape traffic based on information from the browser.

Baseline analysis was conducted by performing a DNS lookup for each site of the Alexa 500 [131]. These websites were chosen because they were the most frequently used. The returned IP addresses of many different DNS providers were compared until a group of four that returned the same set of addresses in the same order was identified. This ensured that there would be no latency issues with the IP addresses. It was also confirmed that the DNS providers supported DNS, DoT, and DoH with the same set of addresses. Next, the application-specific DNS framework was leveraged to determine the overhead provided. Four different applications (Firefox, curl, wget, nslookup) were run, and the overhead of the application-specific DNS method was measured against the normal system DNS, with each application being assigned to one of the chosen DNS providers. Each application was assigned a unique DNS server, and the chosen Alexa 500 sites were queried, while both the time and the CPU overhead of the query were measured.

7.3.2 Performance Experiment Results

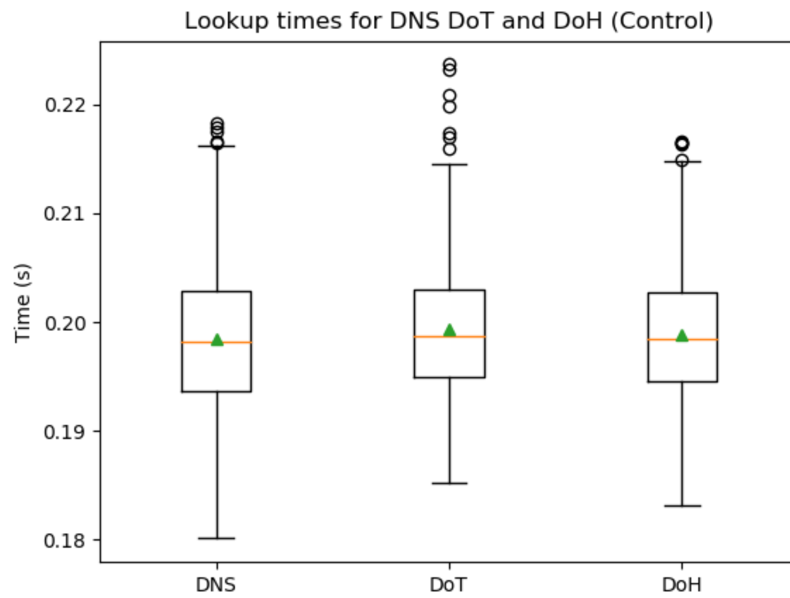
As presented in Figure 7.3, the eBPF solution does not substantially impact the overhead of a DNS connection, adding only 0.44% to the time it takes for a normal lookup. Meanwhile, DoT has an overhead of an additional 8.15%. It was verified that each request is sent to the correct new DNS server by comparing the returned address with a DNS query from the virtual switch. The time overhead was analyzed by comparing the lookup times for all of the Alexa 500 sites with and without the eBPF solutions. The experiment was repeated 20 times to get a clustered grouping of results for comparison.

While it was not possible to directly alter the DoH packets, the CPU overhead was measured for monitoring the calls and returned values as an additional 3.13%. It is shown that eBPF is a valid method for securing privacy by spreading DNS server traffic and limiting tracking. The number of cycles used by the four applications was measured to compute the CPU overhead, both with the eBPF uProbes monitoring and without. From there, the number of additional cycles that were consumed by the monitoring was calculated. Similar to the above, the experiment was repeated 20 times to get a consistent clustering. While the DoH solution does take less time than the DoT solution, it is important to remember that the DoH monitoring is a passive monitor, not an active redirection. A full DoH redirection would require more overhead than DoT due to the extra overhead of the HTTP connection.

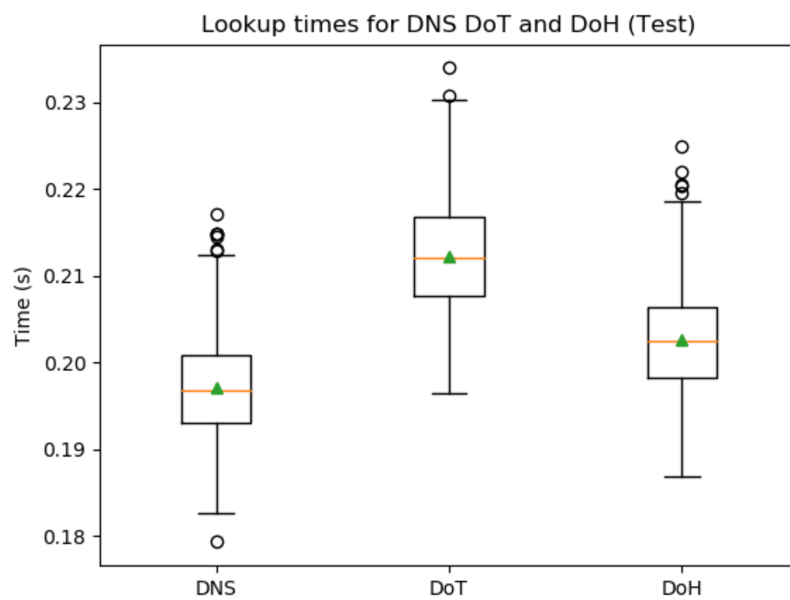
7.3.3 Privacy Experiment Scenarios

Three types of experiments were performed to evaluate if the eBPF implementation protects the user's privacy. For the first type, the attacker was modeled as a user on the same system who could send DNS queries of their own, monitor traffic amounts, and even run WireShark in an unprivileged manner. This was found to be an adequate approximation for attacks on shared computing resources, such as cloud computing and similar systems [60][7].

For the second type, an eavesdropper was placed on the network to disrupt DNS queries and monitor all queries from this selected system. It is shown that the eBPF for DoT and eBPF for DoH systems described in Section IV prohibit information leakage and uphold user privacy.



a) No eBPF running



(b) Using eBPF

Figure 7.3: DNS lookup time measures of the Alexa top 500. Minimal overhead is observed, indicating good performance.

The third type is based on Herrmann et al. [34], where bits of information exposed to a DNS server based on applications running and sites visited are calculated. It demonstrates how DNS servers can build a user profile and what information is most valuable. Using the 1BD-improved described in Section IV , the eBPF implementation is extended to minimize the number of bits sent to any server from a list of viable options. The number of bits exposed in real-time was calculated for DNS and DoT. Although it was not possible to change the packets nor improve privacy, it was possible to expose what is being leaked and flag any potential anomaly for the user. It is shown that the number of bits exposed is minimized based on the number of available servers and that a user running this system has better privacy than a user without it. Further, this system is compared with the test implementation of EncDNS to compare both information exposure and overhead.

7.3.4 Privacy Experiment Setup

The same structural components of the virtual environment used to evaluate performance were used to analyze user privacy. A private network was established through the use of Ryu [89] in a simulated environment. The DoH server and four DoT/DNS servers were simulated, rather than chosen from active servers, and connected to a virtual machine. Within this network, hooks were created for all four groups of potential privacy threats (Section 7.1). Instead of importing all of the Alexa 500 sites, the 50 most visited sites were loaded on the simulated name servers. Each of the simulated name servers had a cache of 15 addresses set to be served in approximately 0.03 seconds, as compared to an un-cached response of 0.1 simulated seconds [157][10]. Addresses were cycled in-and-out of the cache in a FIFO queue. All three eBPF tools (eBPF for DNS, eBPF for DoT, eBPF for DoH) were run on the test system separately and continuously analyzed the information shared with each attacker. The virtual machine had two users: an experimental user and an attacker user. The experimental user followed a set pattern of queries through the Alexa top 50 sites. The attacking user was set to continuously query the core name servers for the top 50 websites and measure response time.

For each potential source of information exposure, a set of metrics was utilized to con-

sider when information would be leaked to a theoretical attacker. For the network eavesdropper, any domain name it is capable of reading is considered exposed. For the server eavesdropper, every time the attacking user predicted a site that was present in the cache of the experimental users' queries, it was considered a leaked site. For the DoH server, as it was not possible to directly edit the information, 1BD-improved and the EFF's Panopticlick [140] was used instead to measure the bits and compare them with the local calculation derived from Hermann et al. [34]. Finally, bit counters were placed on each of the four DNS servers for the ISP simulation, which incremented every time a new URL was requested from the experimental user. Once a counter reaches 30 unique addresses, the user is considered deanonymized [34].

7.3.5 Privacy Experiment Results

Regarding the first scenario, the network eavesdropper, it was found that cleartext was compromised and that both DoT and DoH performed as expected by keeping the DNS transactions private through encryption. EncDNS was easily capable of defeating this, which is unsurprising given that this is a simple attack. In the second scenario, the shared system attacker had the same win condition. Every address it determined the user was querying was considered successful in breaching a users' privacy. It succeeded at a higher rate than the network eavesdropper and was capable of defeating the DoT implementation due to cache linkage over the shared stub resolver. However, both the EncDNS privacy tool, and this method was capable of defeating the shared system attacker by spreading DNS queries over multiple systems.

The third experiment revealed issues with the traditional DoT and DoH approaches. For DoH privacy, a simple running comparison of the 'calculated' bits revealed and 'actual' bits revealed was conducted on the server-side. The 'actual' bits revealed were calculated using a 'truth' reference provided by EFF. It was discovered that EncDNS underestimates the amount of information revealed due to not taking into account the browser user-agent string. In this case, eBPF provided the user with the number of leaked bits and updated information on the user-agent string, information not provided by other tools. This provides the user with

more information about potential threats to anonymity, a vital concern for many users.

Regarding scenarios addressing the ISP, it was determined that EncDNS leaks considerably more information even if it is set to do a parallel resolution. It does not take into account what information the server already knows, and thus, over extended periods, it will eventually leak all information about the user to the ISP. This is a significant threat to user privacy, as allowing the ISP to access all user information is of utmost concern. With *ROS-FM*, eBPF is aware of how much information it has given to each server and can load balance the bits of information revealed to each resolver. The bits revealed were determined to be minimal, protecting the user.

7.3.6 Discussion

The results are summarized in Table 7.1. The tool provides the user with thorough protection in all cases. Using this solution, it is clear that *ROS-FM* allows users to quickly and efficiently protect users from data leakage and privacy attacks. The tool provides an avenue for users to change which DNS server they are using continuously or to sandbox applications into their unique DNS server. This tool provides users with greater control over their data and provides a user-friendly method for privacy-conscious users. While DoH appears to protect most cases, it is crucial to consider the extreme vulnerabilities posed by the browser vendor not addressed by DoH. Using this tool, the user is protected in all cases with DoT and provided support for DoH in regards to the browser vendor by receiving a complete analysis of their privacy and ensuring that data leakage only occurs to the chosen DoH provider.

Additionally, through experimental evaluation, it was determined that the browser could leak an arbitrary amount of information using the user-agent string. Since the browser controls the user-agent string, an arbitrary amount of data may be leaked to the vendor. An adequate action suggested for this type of attack is to warn the user if their identification changes for any reason. Such warnings are provided to the user by this eBPF implementation. It is demonstrated that such a solution does not impact application performance. Given that DNS providers now have the technological capabilities to extract specific user information from DNS traffic, it is believed that users will be well served by this method, allowing

users to shift their DNS providers regularly. This system outperformed the DNS privacy tool, EncDNS, in privacy against the ISP and browser vendor, with similar behavior for more common eavesdropping threats. Ultimately, it comes down to user preference and trust level, but this method provides a far greater amount of control to the user with minimal performance overhead.

Threat	Attack Strategy	DNS	DoT	DoH	EncDNS	eBPF + DoT	eBPF + DoH
ISP	De-anonymization	Vulnerable	Vulnerable	N/A	Partially Protected	**Protected**	N/A
	DNS Query Leakage	Vulnerable	Only if Allowed	Protected	Protected	Only is Allowed	Protected
	Malicious Resolver	Vulnerable	Vulnerable	Protected	Vulnerable	**Partially Protected**	Protected
Browser Vendor	De-anonymization	N/A	N/A	Vulnerable	Vulnerable	N/A	**User Given Warnings**
	Man-in-the-Middle	N/A	N/A	Vulnerable	Protected	N/A	**Protected with User Support**
Network Eavesdroppers	De-anonymization	Vulnerable	Protected	Protected	Protected	Protected	Protected
	DNS Query Leakage	Vulnerable	Protected	Protected	Protected	Protected	Protected
	Man-in-the-Middle	Vulnerable	Protected	Protected	Protected	Protected	Protected
	Malicious Resolver	Vulnerable	Vulnerable	Protected	Partially Protected	**Protected**	Protected
	DNS spoofing	Vulnerable	Vulnerable	Protected	Protected	**Protected**	Protected
Server Eavesdroppers	De-anonymization	Vulnerable	Protected	Protected	Vulnerable	Protected	Protected
	DNS Query Leakage	Vulnerable	Vulnerable	Protected	Vulnerable	**Protected**	Protected
	Man-in-the-Middle	Vulnerable	Protected	Protected	Protected	Protected	Protected
	Malicious Resolver	Vulnerable	Vulnerable	Protected	Partially Protected	**Protected**	Protected
	DNS spoofing	Vulnerable	Vulnerable	Protected	Protected	**Protected**	Protected

Table 7.1: Evaluation of security measures for DoT and DoH with and without the eBPF solution. Only attacks with vulnerabilities to each threat are included.

7.4 Conclusion and future work

7.4.1 Conclusion

In this chapter, a use-case applying *ROS-FM* to help provide DNS privacy to users, though the use of eBPF, is demonstrated. Using this method, users can control their DNS traffic at the application level, which is currently a slow and time-consuming process using other methods. It is demonstrated that it can be done without heavily impacting the performance of the applications. While eBPF cannot interact with encrypted communications such as DoH invisibly, this tool instead provides monitoring and verification of the contents of the lookup. For privacy-conscious users, we recommended that they regularly change their DoH providers to lessen the impact of fingerprinting.

7.4.2 Future Work

In the future, we plan to expand this eBPF framework across additional DNS options. Of chief interest is in providing users with more control over their DNS queries. We hope to develop new techniques to analyze encrypted DNS traffic before transmission and potentially provide both filtering and alterations. Additionally, we aim to allow network administrators to recreate the functionality currently provided by passive DNS in a DoH and DoT filled world without compromising user privacy.

A question worth considering is at which point conscious privacy monitoring becomes too extensive, as there are many benefits to ISP DNS monitoring, such as network shaping and pre-fetching. While increasing user privacy is vital, a completely private internet would probably not work as well and may not be as enjoyable for the user. Awareness of the long-term impact should be considered when creating such tools for increased privacy and maintaining a coherent system.

Part III

External System Verification

Chapter 8

State Encoding

"To be sure of hitting the target, shoot first, and call whatever you hit the target."

Ashleigh Brilliant

8.1 Introduction

Thus far, this dissertation has discussed software tools to protect ROS systems from internal attackers. However, this leaves quite a large area of unaccounted for external attacks. Of particular interest here are attacks that appear legitimate to software but leverage the real world to damage the robot. For example, an attacker could saturate a sensor and render a robot' blind,' potentially causing real-world damage as the robot loses awareness of its surroundings. This is a unique feature of cyber-physical systems, where, unlike traditional software, the state of the world is an additional concern, and data may be good or bad depending on an unobservable state. External attacks are notoriously difficult to defend against and are invisible to traditional firewalls as the sensor is functioning 'correctly' and will not raise a flag. In this chapter, a solution for anomaly detection for sensor spoofing attacks is introduced. This tool monitors the published sensor data in ROS-Immunity and works in conjunction with ROS-FM to address both internal and external attacks.

As discussed previously, robots are cyber-physical systems designed to perform spe-

cific tasks and ease human work. With robots, the physical world is highly coupled with cyberspace. Firstly, sensors perceive the physical environment; then, the control software chooses actions, potentially in collaboration with other cyberspace agents (*e.g.*, other robots or the cloud); finally, actuators perform those actions on the physical environment. Regardless of their capabilities and size, robots take their actions based on what they sense.

Thus, robots are targeted by sensor spoofing attacks that can force an incorrect behavior in the robotic system and undermine the success and safety of critical operations[55]. In this chapter, the focus is placed on Light Detection and Ranging (LiDAR) systems. These systems are primarily employed to achieve high positioning resolution, in substitution or support of other approaches that locate the robot in indoor environments [179]. Any alteration of these sensor data can silently force the robot to initiate dangerous maneuvers for itself and the environment in which it operates. Regardless of the robot's purpose, sensor quality and robustness are highly requested to perform eventual mission- and safety-critical operations.

This chapter proposes a novel anomaly detection approach for sensor spoofing attacks to detect anomalies in the LiDAR data. It is based on an autoencoder architecture, an artificial neural network that aims to learn efficient data representation in an unsupervised fashion [205]. This approach is predicated on the concept that the physical world is continuous and consistent. Stepping into the real world, laws govern movement and physical behaviors, not present in a purely digital world. These constraints may be detected in time-series data and, therefore, can serve as a beacon for anomaly detection. This constrained, rule-based approach is precisely the strength of artificial neural networks.

8.2 Spoofing Attacks Model

In this section, the attack model that targets a single sensor system of the robot is outlined. This chapter focuses on the LiDAR sensor system as it is a common sensor type for robotics whose security is underexplored.

8.2.1 Independent Assumptions

The following assumptions are held for this chapter:

- It is assumed that the robot will always start in an uncompromised state.

- The robot's environment is not prone to sudden changes, such as objects appearing close to the robot without crossing the intermediate space.
- It is assumed that the attacker can alter the sensor input state of the LiDAR system either through the associated software, directly interfering with the hardware, or broadcasting their LiDAR signals toward the receiver.
- It is assumed that while the attacker can arbitrarily control the value of the LiDAR , the robot does not start under the attacker's control and that the robot has access to an unaltered LiDAR input.

8.2.2 Potential Spoofing Attacks

Ten potential spoofing attacks were identified from both the state-of-the-art and the previous research discussed in this dissertation, with these assumptions in mind. Ten attacks were formalized in the following subsections.

Percentage Spoofing Attack

A percentage spoofing attack is one where all input data is shifted by a set percentage, multiplicatively changing all values with larger values more affected than smaller values. This has the effect of causing the world to grow or shrink around the target, leading to collisions with objects that could not be detected and trapping the target with its collision avoidance system if it perceives gaps that it could fit through as too small. Building off of previous work[65], a 10% change to all values is used, as it is the smallest percentage difference that can both ensure a robot collision and trap a robot with the default route planning software.

Value Spoofing Attack

Instead of spoofing by a percentage of the current returned value, the value spoofing attack shifted up or down the points by a set value of 5 centimeters (~25% of the robots total size), introducing a predictable bias into the system that all objects are either nearer or further away. Unlike percentage bias, this is more difficult to detect as it more closely mimics the effect of noise, a biased sensor, or bug instead of a rapid shift since all values are changed by the same amount.

Rotation Spoofing Attack

A rotation spoofing attack does not seek to change any of the values measured by the LiDAR sensor, instead of changing the indices where they are stored. Since these indices are the angles for the measurements, the effect is a rotation of the environment around the robot, such that the direction it thinks it is facing is no longer the direction it is truly facing. This causes a disorientation effect where the robot's predicted plan of motion does not directly map to the robot's surroundings. For example, the robot might try to turn right when it should be going straight.

Zero Replacement Spoofing Attack

A zero replacement spoofing attack simply replaces all inputs to a value of 0. To the robot, this appears as though it is completely blinded for the LiDAR or that there are objects completely blinding its LiDAR sensor. This effectively removes the robot's LiDAR sensor from consideration, and if it is the robot's primary means of sensing, then the robot cannot continue acting safely. Given that the LiDAR has a minimum range for measurement, which it returns if it runs into an error, this should be very easy to detect.

NaN Replacement Spoofing Attack

The NaN replacement spoofing attack replaces all values with a value of NaN (not a number). Unlike a Zero Replacement attack, this can occur naturally if the robot is somewhere where the nearest obstacle is beyond its sensor range. As such, the robot considers itself completely alone in its environment and can act with impunity as far as safety checks are concerned. In any scenario where there are obstacles in its path, the robot cannot detect or avoid them in this case.

Repeated Data Spoofing Attack

The repeated data attack takes the last input to the robot and continuously replays it as the only new input with the addition of Gaussian noise similar to the generic sensor noise. This effectively means that it is the last unique input it receives, giving the robot the illusion that its environment is moving with it like if it were on a 'treadmill.' Due to this, the robot cannot react

to its movements and thus should stop, but that is not usually the case. The sensors report that the robot is stopped, but in reality, it could still be moving. This attack is exceedingly difficult to detect without relying on other inputs as it is entirely functionally identical to the robot coming to a stop.

Window Repeated Data Spoofing Attack

In a window repeated data spoofing attack, the data is repeated as a sliding window instead of having just one value repeated continuously. This creates a noticeable *lag* effect where updates to the robot's sensor happen far desynchronized in time as compared to reality. This has a very damaging effect on control loops as they attempt to compensate for the lag, creating more jerky and aggressive behavior in the robot.

Sector Value Spoofing Attack

Sector value spoofing attack aims to closely follow practical external attacks, where the attacker is not directly modifying the sensor data but instead is broadcasting their fake LiDAR signal from a direction. For example, they may replay the signal with a time delay, mimicking the value spoof attack only on part of the data. These attacks only affect the part of the input data while leaving the rest untouched.

Real World Spoofing Attack

The real world spoofing attack assumes that a LiDAR spoofer is a discrete object located near the robot that can only attack the parts of the LiDAR input that directly pass near it. Like a sector value spoofing attack, it only affects the part of the data, but the part of the data changes as the robot moves around the data. This spoofer appears more plausible to what the sensors would detect in a real-world context, such as an attacker spoofing on the side of the road to interfere with a self-driving car. This attack is based on the information outlined in [55]. It is hard to detect as it rapidly varies in time and changes, making it hard for the defenses to detect.

Frog Boiling Spoofing Attack

Named after the commonly known attack[181], the frog boiling spoofing attack attempts to avoid detection by only subtly changing the input at any point below the detection filter. Over time, these small changes add up to a substantial change in the input, but it is hard for the defender to see the change at any point. The most commonly known example of the sensor attack in the wild is the case where a yacht was driven hundreds of kilometers off course through a GPS sensor attack where every new position was less than 10 centimeters off of the last one, appearing that it was on course when in reality it was not[67].

8.3 Architecture & Approach

8.3.1 Architecture

The architecture of the detector is shown in Fig. 8.1. The detector is implemented as an LSTM autoencoder, and it is split into two parts: an encoder and a decoder. The encoder is designed to create a compressed representation of the input data, while the decoder is designed to recreate the original input. Before detailing the architecture, a short introduction to autoencoders and decoders is given in the following: autoencoders are a special type of Neural Networks used for reconstruction purposes [205]. The main idea is that it is trained by taking an input x, and given an output objective y which is identical to the input (i.e., $x = y$), the goal is to reconstruct the input as similarly as possible by minimizing the difference between both. It is composed of two parts, an encoder, which takes the input x and encodes it to match a learned representation z, called *latent space*, and a decoder that tries to decode the representation and determine the original input. The latent space's dimension is smaller than the input's, which necessarily means loss of information [205] [97]. Thus, the autoencoder's most important work is to proceed to dimensionality reduction while keeping the most critical information that allows a nearly lossless reconstruction, or at least, minimize the loss. Autoencoders are considered a type of self-supervised learning model that can learn a compressed representation of input data.

Long Short-Term Memory (LSTM) [188], is a particular type of neural networks used in

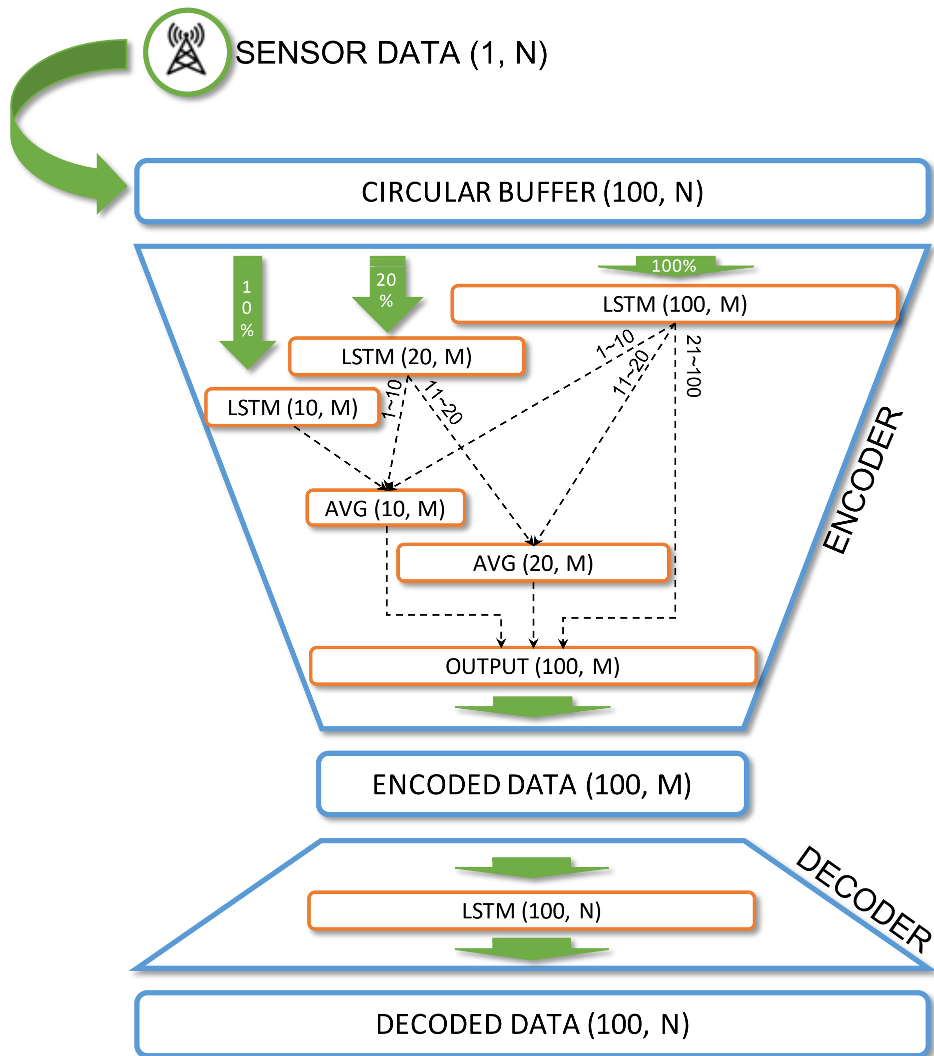


Figure 8.1: Architecture of the encoder and decoder.

deep learning. An LSTM autoencoder is an implementation of an autoencoder for sequence data using an encoder-decoder LSTM architecture. The encoder can be used to compress or encode sequence data that may be used as a feature vector input to a supervised or unsupervised learning model or in data visualizations. Given a sequence dataset, an LSTM model is configured to read the input sequence, encode it, decode it, and recreate it. The evaluation and the performance of the model are based on the model's ability to recreate the input sequence. Among the advantages of LSTM regarding the classical neural network are that LSTM has cells that remember their inputs over a long period. It means that a long sequence of time with its data dependency is considered in the training phase.

In this case, the autoencoder's input is a circular buffer with shape 100xN floating-point values, where N is the sensor input dimension. This introduces a temporal correlation to the input allowing a memory of previous states. At each timestep, the buffer is fed another LiDAR measurement as a vector of N points, and the oldest value is removed. The input layer distributes the data over three LSTM cells to better detect spoofing attacks. The first cell works on only the past ten samples and is the primary line of detection for the most straightforward attacks, which detection can be performed earlier. The second cell is trained on the past 20 samples and serves as the primary check for potential attacks. Since each attack effectively alters the input to the first layer twice as much as the second, this provides a precise measure of potential attacks. The last cell looks at the past 100 samples and is mostly used to detect slow or subtle attacks, such as Frog Boiling attacks.

The outputs of the LSTMs are merged in a stepwise manner; the overlapping parts of each output are merged, and the unique parts are appended. This means that the output of the first layer, the first ten outputs of the second layer, and the first ten outputs of the last layer are averaged and make up the first ten components of the encoded output. Similarly, the second 10 outputs of the second layer and the second 10 outputs of the last layer are averaged to make up the next ten components. The final 80 components are directly taken from the output layer. The output length of the LSTM cells can be tuned to alter the compression ratio, and it is defined as M, where $M<N$.

The decoder component has its own LSTM cell, and it is trained to reverse the result

from the encoder value and recreate the initial input. This last LSTM cell's output length is *N*, the same dimensionality of the autoencoder input.

A necessary metric for this solution is the reconstruction error. The reconstruction error is a measure of how successful the autoencoder is at recreating the input once it has been encoded and decoded again. It is defined as:

$$reconstruction_error = \Sigma(x - y)^2$$

where x is the autoencoder input and y is its output.

The reconstruction error of the autoencoder is measured to calculate the amount of noise in the system. Whenever the amount of noise vastly changes in the system, it is a strong indicator that there is now spoofing. The system is structured to strongly prefer false negatives to false positives. Unlike Sakurada *et al.*[199], the data is split into three separate LSTMs to provide a weighting of the input allowing for better pinpointing of the beginning of the attack.

8.3.2 Learning Phase

In the initial phase, the models were trained over an attack-free dataset to determine typical LiDAR data benchmarks. The model was provided with examples of several situations and allowed to learn the best way to represent the data in an unsupervised fashion. To train the neural network, the training data was read in and reshaped into a rolling set of 100 sample buffers offset by five samples each time, *i.e.*, the first buffer was sampled 1-100 while the second was sample 6-105. Once the model has been fully trained, the dataset is fed into it through the normal circular buffer one sample at a time. The decoded output was recorded and used to calculate the reconstruction error for each time step of the training data. The mean μ and standard deviation σ of the reconstruction error are calculated and stored alongside the compression phase model.

8.3.3 Detection Phase

For the anomaly detection phase, the detector uses the encoded data and decodes them to reconstruct the original input. Then, it computes the reconstruction error and compares it to the previously computed mean μ. If the current reconstruction error differs from the mean μ more than $\pm 3\sigma$ [69], the alert of a potential attack is raised, returning the sample wherein it is was detected. The 3σ threshold is used to strongly discourage false positives in the calculations. False positives are a common source of issues within security systems, and it is vital to limit this issue.[94]

8.4 Experimental Results

8.4.1 Setup

The setup consists of a Linux Virtual Machine (Ubuntu 16.04) with 4 CPUs and 8 GiB of memory. Qemu emulates the VM in a Linux Server equipped with Intel Xeon CPU E5-4650 v4 @2.20 GHz. The following were used: GAZEBO version 7.0.0 [190] for the simulation of robots and environments, ROS Kinetic Kame in the emulated robots, with the solutions implemented using the Python 3, Tensor Flow version 1.13.1 [201], and Keras version 2.2.4 [43] frameworks. The simulated robot is a Kobuki, equipped with an RPlidAR A2M8 as the LiDAR sensor. This sensor has a frequency of 6 Hz, *i.e.*it reads 6 samples every second. A sample is a vector of 720 floating points that are the measurements from 720 directions around the robots, distributed in 8 sectors. Each point contains a value between 0 and 15 meters. This value is the distance from the robot to the closest object in that direction. If the object is further than 15 meters, the value is set to NaN. The experiment is run using data collected from two different simulations built using the `rrt_exploration` packages for multi_robots[90]. In both simulations, five robots are tasked to explore the environment to build a shared common map without any truth reference. Thus, they rely on shared data to reconstruct the map. They differ in the simulated environments that the robots have to explore: the first is a one-story house, while the second is a larger maze.

8.4.2 Dataset Collection

The dataset is collected from the Gazebo simulations and stored in ROSbag files [198]. The data collection began when the robots started moving and concluded when they were confident that they had fully explored the area and correctly merged the map. The dataset contains many unique geometries for the LiDAR to record and multiple moving objects crossing in and out of vision.

Before training the model, the dataset was normalized to avoid any issue that could confuse or incorrectly train the model. *NaN* values, *i.e.*nothing identified by the LiDAR, must be treated specially. All the points are normalized to a range between 0 and 1, but NaN values are substituted by the value 2 to indicate that the laser did not return and differentiate from actual points.

To inject the attacks into the dataset, a random sample was selected within 100 samples of its midpoint. The random value was chosen to ensure that there was no potential source of predictability for the testing. Once the starting sample was chosen, the attack was applied to the remainder of the data set. A total of 20 datasets were created for testing, two per attack type.

The datasets are publicly available at `https://bit.ly/2Mndt4T`. It is strongly believed that sharing these benchmarks can help the research of robot anomaly detection by comparing different solutions on generic data.

8.4.3 Model Tuning

Four different detectors were initialized by tuning the architectural model. N and M are the two tunable values of the architecture, where N is the dimension of the inputs and M is the compressed data size (*cfr.* §8.3.1). All four detectors work with data read from the same sensor that provides a vector of points every timestep. Thus, N was set to 720. Every detector is characterized by a different value of M: 80, 150, 300, and 600. This provides different compression ratios that can also be computed as **state space saving**, *i.e.*reduction

Dataset	First Simulation				Second Simulation			
Compressed Data Size (from 720)	80	150	300	600	80	150	300	600
State Space Saving	89%	79%	58%	17%	89%	79%	58%	17%
Percentage Spoofing Attack	*ND*	50	25	25	*ND*	75	25	25
Value Spoofing Attack	50	25	15	20	50	25	15	20
Rotation Attack	10	10	5	5	20	10	10	5
Zero Replacement Attack	10	10	10	10	10	10	10	10
NaN Replacement Attack	0	0	0	0	5	5	0	0
Repeated Data Attack	*ND*	*ND*	*ND*	*ND*	*ND*	*ND*	*ND*	*ND*
Repeated Data Window Attack	*ND*	*ND*	45	35	*ND*	45	35	25
Sector Spoofing Attack	10	10	5	5	50	20	20	10
Real World Spoofing Attack	*ND*	20	20	20	*ND*	50	20	20
Frog Boiling Attack	*ND*	*ND*	*ND*	85	*ND*	*ND*	*ND*	90

Table 8.1: Detection Latency in Number of Samples (*ND*: not detected)

in size relative to the uncompressed size. It is defined as

$$space_saving = 1 - \frac{compressed_data}{uncompressed_data} = 1 - \frac{M}{N}$$

and it is expressed as a percentage. For example, the first detector is tuned with an output dimensionality of 80 for the encoder. It means that compressing 720 points to 80 saves the 89% of space for storing the robot state.

The model was trained on the dataset of the first simulation, not compromised by any attack. The automatic split function in Keras was used to split the data 80% for training and 20% for testing, to train the autoencoder to pick up normal LiDAR behavior. Once the model has been trained fully, the data is run through a second time to calculate the mean and standard deviation of the model's reconstruction error and the detection thresholds.

8.4.4 Experimental Results

Eighty (80) experiments were run, one experiment for each dataset, simulation, and model. Results are shown in Table **??**. Latency is defined as the number of samples provided to the detector necessary for detection from the sample in which the attack started. Columns 2 to 6 report the latency for the experiments with the first simulation data, while columns 7

to 10 are for the second simulation. The second and third rows label the columns according to the compressed data size and state space-saving of the used detector.

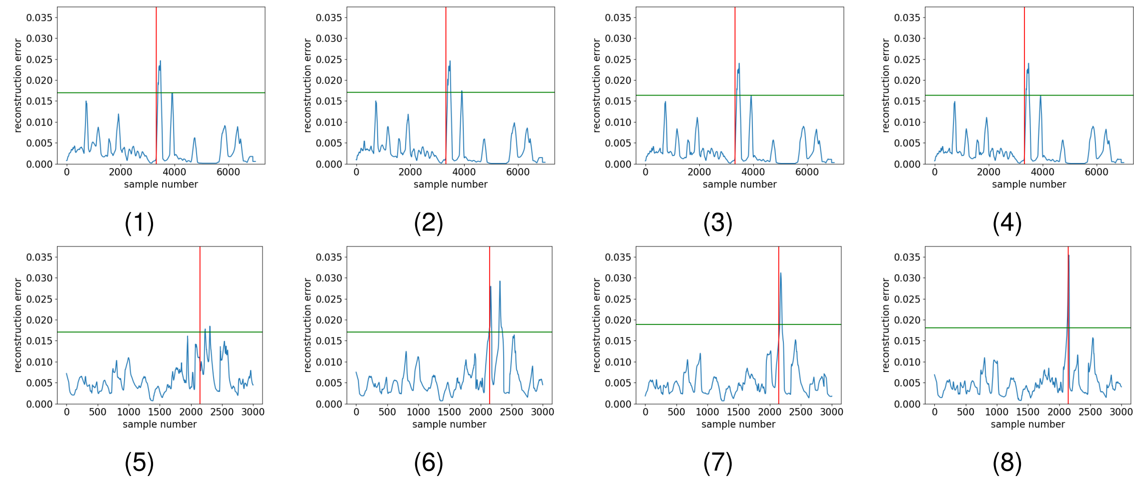


Figure 8.2: Reconstruction error using the data from the first simulation (a,b,c,d) and second simulation (e,f,g,h), and with a space-saving of 89% (a,e), 79% (b,f), 58% (c,g), and 17% (d,h). The horizontal axis is labeled with values that specify the sample number in the time series, while the vertical axis is labeled with construction error values. The red vertical line shows the point where the attack starts, while the green horizontal line indicates the detection threshold. Each pair of space-saving has its threshold for detection.

This solution works well in most situations (59 experiments out of 80), even with a high space-saving, detecting the attacks within an average of 20 samples of latency (3.33 seconds). In particular, this approach is effective with a 99% True Positive rate and a 10% False Negative rate for 17% space-saving, decreasing to a 50% False Negative rate while maintaining the 99% True Positive rate for the 89% space-saving. The True Positive rate is determined statistically and is defined as a reconstruction error above the threshold when there is an attack. The False Negative rate is defined as a reconstruction error below the threshold when there is an attack. **Detection is more effective with 17% space-saving. However, 58% space-saving could handle almost all of the same attacks while using half the data.** Even if the detector was trained on the first simulation environment, it could also detect the same attacks on the new (and more complex) environment of the second simulation. The delay is a maximum of 40 samples (6.66 seconds) between the two envi-

ronments' performances at the highest compression rate and 10 samples (1.66 seconds) for the lowest. Figure 8.2 shows the computed reconstruction error throughout the dataset for the value spoofing attack experiments. The figures clearly show the attack's effect on the reconstruction error with the large spike of noise. The secondary spikes come from the three-layered approach as all parts of the detector react to the attack. Once the attack has filled the buffers, it is no longer 'novel' data, and thus the error drops to normal. This means that once an alert has been raised, the robot can no longer trust its LiDAR sensor until it has been returned to a safe state. The loss of information in the data compression prevents the detector from unveiling a spoofing attack such as percentage, repeated data, real-world, and frog boiling spoofing attacks. In particular, the frog boiling spoofing attack is very challenging to detect, as the difference between any given two attacked samples is less than the difference from the noise between them. The proposed detection mechanism is not able to detect the repeated data attack because using only LiDAR data, they are identical to the robot just stopping during regular operation. This scenario is often repeated within the data set, *e.g.*the beginning of the operation, the end of the operation, and every time the map is updated with new goals, the detector assumes it is normal. This can be solved by adding meta-sensor information (*e.g.*message time or last update) to the detector, training the detector to detect specific implementations of the attack, or adding a new input from another source. If a different part of the robot thinks it is moving and the LiDAR is not changing, something has gone wrong. Overall, this system can detect most attacks on the system and allow for operator intervention.

8.5 Conclusion

In this chapter, an anomaly detection method for detecting sensor spoofing attacks against robots was detailed. This solution is based on an autoencoder neural network that leverages both the spatial and temporal features of the sensor data in order to reconstruct the robot's sensor data and detect anomalies. The experimental results highlight the ability of this method to detect attacks. As of 2020, this is the first work that faced the challenge of sensor data anomaly detection through autoencoders by also enabling efficient robot state

representation.

This work also has some limitations: usage of simulated data and limited experimentation. While there is confidence in Gazebo's high quality of data, it is understandable that experimentation on-the-field will be more valuable.

Autoencoders have been applied to secure against anomalous inputs from sensor attacks in relevant related work, which we briefly summarize next. Sakurada *et al.*[199] were the first to propose using the reconstruction error of an autoencoder for anomaly detection. Similarly, Chong *et al.*[69] proposed using a deep learning approach to extract features from video frames. The autoencoders they presented are built with a Convolutional Long Short-term Memory (ConvLSTM) model, a variant of the LSTM [188] architecture. They experimented with their approach on avenue, subway, and pedestrians video datasets. Medel *et al.*[59] proposed a similar autoencoder solution by using ConvLSTM model on surveillance videos.

8.6 Key Contributions to ROS-Immunity

The autoencoder framework proposed here is a key component in ROS-Immunity. It is re-implemented as a plug-in for *ROS-FM* and extended to support generic sensor data. The tool is most tested and optimized for LiDAR data; however, the extension provides flexibility for ROS users to protect against attacks on most sensor types. This approach was demonstrated on a wider class of sensors in Lagraa *et al.*[152] and Amrouche, 2020 [162].

In *ROS-Immunity*, this tool protects against external attacks and sensor anomalies. Robots on the network utilize the encoding reconstruction error to monitor neighboring robots for signs of compromise. This is a vital inclusion in the *ROS-Immunity* framework as the primary defense against these attacks.

Part IV

Automated Vulnerability Detection

Chapter 9

ROSploit

"One must not look down on tricks that deceive only fools, my son, as the better part of the people of the world are patently foolish."

Anonymous

9.1 Introduction

In the deployments of robots, it is vital to consider their safety-critical nature. A faulty robot can irreversibly damage the physical environment in which it operates, including being harmful to human beings. Thus, testing the robustness of robotic systems is crucial to ensure safety.

Automatic vulnerably scanning is the process of checking a system for the presence of any known vulnerabilities, with limited user intervention. It is the bare minimum for security work. Without it, users cannot accurately gauge if their systems are configured correctly or have the ability to address known threats correctly. Currently, the most well known automatic vulnerability scanners are Metasploit [191], and Nmap [184], which primarily address normal computing processes. However, few solutions exist for ROS systems, the following being one of the first of its kind published.

Given the unique structure of ROS systems, a network scanner is the first logical step to creating an automatic vulnerability scanner for ROS. This chapter discusses *ROSploit*,

a set of tools for analyzing the security of ROS systems. *ROSploit* allows researchers to scan for potential security vulnerabilities quickly. Regular scanning is critical to ensure that a system is protected, and easy-to-configure automatic scripts are the best way to ensure that happens. This is even more important in robotic systems, as vulnerabilities can quickly impact real-world functionality and quick response is of utmost importance. Additionally, *ROSploit* supports modular scripting, which is a crucial contribution to ROS systems.

This chapter demonstrates the functionality of *ROSploit* through several experiments designed to demonstrate the tool's full scope on known vulnerabilities and flaws in ROS. When published, *ROSploit* supported all known vulnerabilities. As time has progressed, it has been continuously updated to support newly discovered vulnerabilities and issues.

9.2 ROS Framework

9.2.1 Threat Model

Similar to previous chapters, a threat model is established. This threat model was based on the attack vectors discussed in Security for the Robot Operating System [72], mainly focusing on the following attacks: Unauthorized publishing, Unauthorized data access, and a packet flooding DOS as a first pass analysis into the security of ROS. For this chapter, these attacks are extended into broader categories, and other attacks that may be possible for those categories are explored.

The *ROSploit* threat model is built upon the industry-standard triad of Confidentiality, Integrity, and Availability(CIA), with an additional secondary focus on the difference in capabilities between internal and external threats. For this threat model, two distinct types of threats, internal and external, and their primary risks were considered. Internal threats are defined as attackers who have access to the underlying Linux system, with access to spawn new nodes onto the system legitimately and have partial network access. Insider threats can come in a wide variety of forms, including; malicious manufactures, compromised nodes [33], disgruntled developers with access,[39] or even simple software bugs. External threats cover a much wider scope. External threats are defined as any threat that does not have partial access to the underlying Linux system. This means that they could not

launch new nodes on their own and that they might not be able to access internal networks. By referencing similar threat models for normal Internet-connected systems[35] , external threats will likely be the vast majority of threats that ROS systems will face.

When analyzing external threats, several areas of known potential vulnerabilities for the ROS system were discovered. These include sensor input[93], tampering, as well as tampering with available IP flows[72]. Any attacker connected to a shared network with a ROS system can perform fake data injections and perform regular IP-scope attacks such as DDOS and MiTM on typical connections. Additionally, given the existence of remote parameter control for ROS services, an attacker may escalate their attacks to remote code execution. This would allow the attacker to load malware on the robot, which would elevate the attacker to be an internal threat instead of an external. Focus is placed on implementing and detecting threats available to an attacker who can communicate with the robot over IP with a secondary focus on insider threat detection.

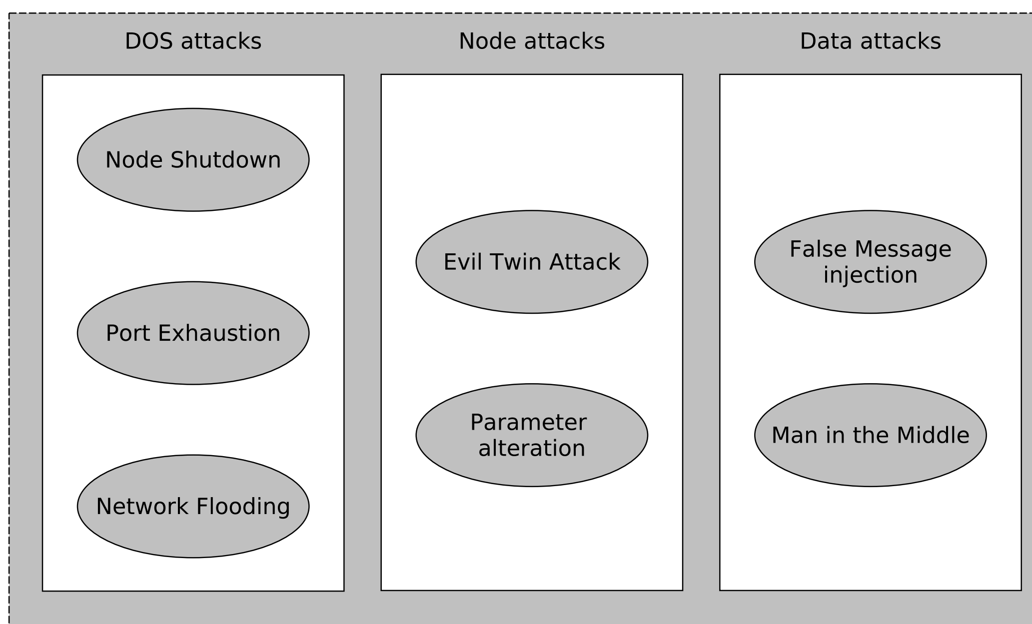


Figure 9.1: A graphical representation of the threat model.

9.2.2 ROS attacks

Attacks against a ROS system can be broadly divided into three categories; DOS attacks, Node manipulation, and Data manipulation. Each of these categories has a broad scope of possible attack vectors and potential effects on the system. The value these categories provide focus on which areas of the ROS system must be hardened and the broad similarities in defense that each category shares. Each category's name was based on the ROS component most effect, though each category also directly corresponds to the CIA triad.

DOS attacks act by denying resources to the system in a way that damages or alters regular operation but does not directly change the system functions. All DOS attacks are against the Availably of part or all of a ROS system. Many of these attacks are identical to regular IOT attacks, with ROS extending the attack surface through individual nodes' potential to be shut down. The DOS attacks discussed in this chapter are detailed in Table 9.1.

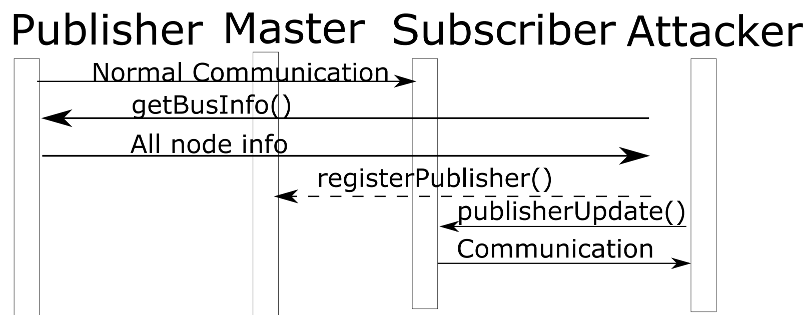


Figure 9.2: A sequence diagram of a potential Evil twin attack against ROS. The dotted line is optional as the attacker does not have to inform the master of the new version of the publisher unless they wish to intercept communication from multiple nodes.

Node manipulation attacks directly target the ROS nodes. All Node manipulation attacks affect the integrity of the robotic system. These attacks change the internal structure of the robot in order to alter the robot's behavior. Node manipulation includes changing which topics nodes are connected to and attacks against the master and parameter servers. The Node attacks focused on in this chapter are detailed in Table 9.3.

Data manipulation attacks include any attacks that alter data as it passes through the

system. Data manipulation attacks include the injection of false data onto a topic, MiTM attacks, and sensor attacks [92]. These attacks are the most difficult to detect, as they can perfectly mimic normal system behavior. The altered data is challenging to detect without monitoring all data traveling through the system. Data manipulation attacks focused on in this chapter are detailed in Table 9.2.

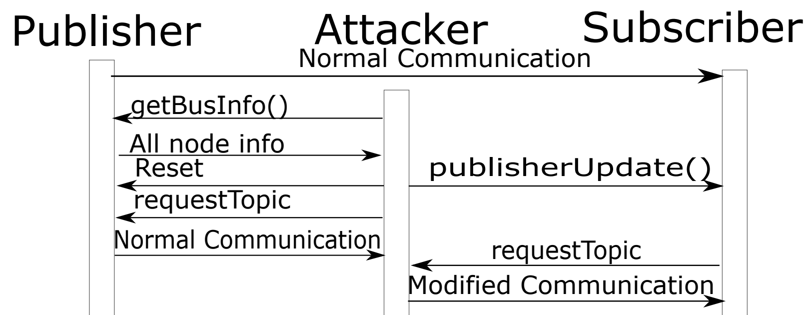


Figure 9.3: A sequence diagram of a MiTM attack. The attack has to be careful with timing, as they need to make sure the subscriber node only subscribes to the attacker's node after the publisher is reset.

9.2.3 ROSploit

ROSploit was designed to assist security researchers in their analysis of ROS systems. This system can be split into two separate components: reconnaissance and exploitation.

The reconnaissance components of the system integrate with the existing research tool NMAP[184] as a set of NSE[14] scripts that can be enabled when scanning a suspected ROS system. As of 2020, there are two main high-level scripts implemented as part of this system. The first script is a master node scan, which focuses on pulling information from the ROS master, while the second is an addition to the normal NMAP comprehensive port scan, allowing NMAP to identify various ROS nodes during the port scan.

The master scan script runs only for the master port(11311) of ROS. This scan calls the *getSystemState()* function, a part of every ROS system. Once it calls the function, it is given a list of every single running node topic and service on the ROS system, which it then parses for further user examination. The return from *getSystemState* is a list of lists, where the first list contains all of the publishers of topics on the system, the second list contains all

Attack Name	**How Functions**	**Why threat**
Node Shutdown	All nodes have an XML-RPC shutdown command available by default. Calling this command will shutdown the node.	If an attacker manages to disable a critical node during normal operation, the system might continue running in a compromised state.
Port Exhaustion	Every time a new node subscribes to a topic, it opens up many new TCP port connections equal to the number of publishers publishing to that topic. An attacker may cause port exhaustion by subscribing to topics with several publishers	On its own, the threat is mostly a resource denial threat. By itself, port exhaustion does not seriously impede the functioning of the robot. If paired with the resetting of nodes once port exhaustion has been reached, or with a system that regularly has to drop and relaunch nodes, the threat becomes more severe as it blocks the ROS system from performing any communication or error correction.
Network Flooding	As with all IP systems, the ROS system can be attacked with standard network packet flooding attacks	A fully flooded network can no longer pass data between various nodes. Given ROS's modular construction, this means that control nodes and sensor nodes are no longer able to communicate. A well-designed robot should freeze at this point, but it is possible that it would instead perform an unexpected and potentially unsafe action.

Table 9.1: ROS DOS attacks

Attack Name	How Functions	Why threat
False Message Injection	False message injection involves subscribing to a topic and publishing data to it.	No authentication on who is publishing on a node allows an attacker to publish false data to any topic. This false data allows an attacker to manipulate behavior alongside valid real data.
Man in the Middle (MiTM)	A man in the middle attack for ROS is to have an attacker interject itself between a publisher and subscriber of a topic or between a service provider. The attack intercepts messages from the publisher, passes them along to the subscriber, and can alter the content beforehand. Figure 9.3 shows a sequence diagram of a MiTM attack.	This attack is one of the most threatening attacks covered by this chapter. Since the attacker has intercepted all communication into the node, the attacker can effectively control the underlying node's behavior. Not only can the attacker craft their packets to the nodes, but they are also able to selectively filter out information to the node, such as removing a single object from LIDAR sensors.

Table 9.2: ROS data attacks

Attack Name	How Functions	Why threat
Evil Twin	An evil twin attack replaces a node with an attacker-controlled version of that node, with the same name and connections, without alerting the rest of the system about the change. Figure 9.2 shows a sequence diagram of a potential Evil twin attack against ROS.	Most of the threats of an evil twin are similar to those from a MiTM attack. The primary threat that is unique to the evil twin attack is that the attacker has direct control over the apparent behavior of a node and thus can influence the behavior of the rest of the system. If the master is aware of the evil twin, the master automatically shuts the original node down by creating an assumption that the system is still working as expected
Parameter Alteration	As part of the core ROS system, the parameter server holds shared information from all nodes an attacker may access and change any of this data.	An attack could use this to put the system into a debug state. Beyond that, attackers can change shared parameter data between nodes, altering the system state.

Table 9.3: ROS node attacks

of the subscribers to topics the third list contains all of the services available. All of these lists contain the names of the various connected nodes. The wide port scan can be run as a part of any scan of the system. It can determine if an open TCP port is a ROS node, part of ROS master, or a ROS service by sending normal XMLRPC requests to the nodes and monitoring the responses it receives. If it fails to receive a response from the XMLRPC requests, it attempts to send a TCPROS subscription request to the open port. Using the response to this request, the system can determine if the node is a topic or a service is running ROSTCP.

Figure 9.4 demonstrates the node scan portion of ROSploit and compare it with the results of the rosgraph tool in order to demonstrate the effectiveness of the scanning tool. The red highlighted section shows that the system correctly identifies the open ports and the name of the node running. For the topic information, note that each publisher has an open for each topic that it publishes on. The script flags these by describing them as topic ports and then naming the publisher, highlighted in purple). Another exciting component from the script, highlighted in green, is the ROS graph tool being identified by the scanning tool. This indicates that one can effectively scan the whole system, including the debugging add-ons.

The exploitation component of the system is developed in Python as a set of modular exploit components. It is similar to Metasploit [191] in design as it is a modular system of scripts that contains scanned exploits to be run against an already scanned target, and it depends on the reconnaissance half of the system to determine which parameters are needed. To facilitate easy development of various exploits, TCPROS, UDPROS, and the rest of the ROS middleware were reimplemented in Python. This allows the user to run *ROSploit*without having to install ROS fully, and it opens up new potential exploit areas where attacks can target the underlying system by, as an example, sending malformed TCPROS messages. In order to test the exploitation half of the system every attack mentioned in Tables 9.1 9.2 and 9.3 has a script implemented. These scripts were tested against a simple ROS implementation and the Autoware[110] self-driving car system. During initial tests, it was confirmed that an attacker who shares the same network as the ROS system could take control of the system. Additionally, the tool was able to perform live editing of data between ROS sensor nodes.

*ROSploit*is implemented as a library of Python functions that give the developer access to the ROS stack's entirety. The tool has all of the core ROS XMLRPC functions implemented and helper functions to communicate with the TCPROS protocol. All of the ROS attacks are implemented as standard scripts that use a common data structure. To develop a new attack, a user of *ROSploit*can quickly integrate the various library functions into their script. *ROSploit*implements all of the attacks listed in the tables9.1, 9.2, and 9.1as one-off scripts. A user can select which attacks they with to run as command-line arguments. As an example, in order to run a MiTM attack, a user would give the name of the topic and the two nodes that the script. The script would then insert itself into the communication between the two nodes on the selected topic. From there, the user is free to modify the communication as needed.

Figure9.5 shows an image of the experimental setup, which consists of a Turtlebot3 robot[202], as well as a connected control laptop which contained the ROS master and the more processor-intensive components. These two components were connected to a shared network with a third computer running ROSploit, and all of the implemented attacks were tested against the system.

9.2.4 Experimental Evaluation

After designing and implementing *ROSploit*, the system was texted to ensure it could analyze the known vulnerabilities. It was run through several attacks.

- The detection script was run against a real-world robot and confirmation was received on whether it could see everything running using the innate ROS tools.
- The exploit tool was tested against several nodes on a simulated robot using Gazebo. This simulated robot was identical to the Turtlebot3 reference robot, but all data was generated instead of measured.
- Three (3) successful attacks were repeated against the actual test Turtlebot3 (False Message, MiTM, Network Flooding) to confirm that the simulator was effective.

After experimentation, it was confirmed that *ROSploit*was able to emulate all of the pro-

Figure 9.4: An example version of the NMAP port scan compared to the plot generated by the internal ROS graph tool. The highlighted names were included to demonstrate that the nodes were successfully scanned while the rest was omitted due to brevity.

posed attacks.

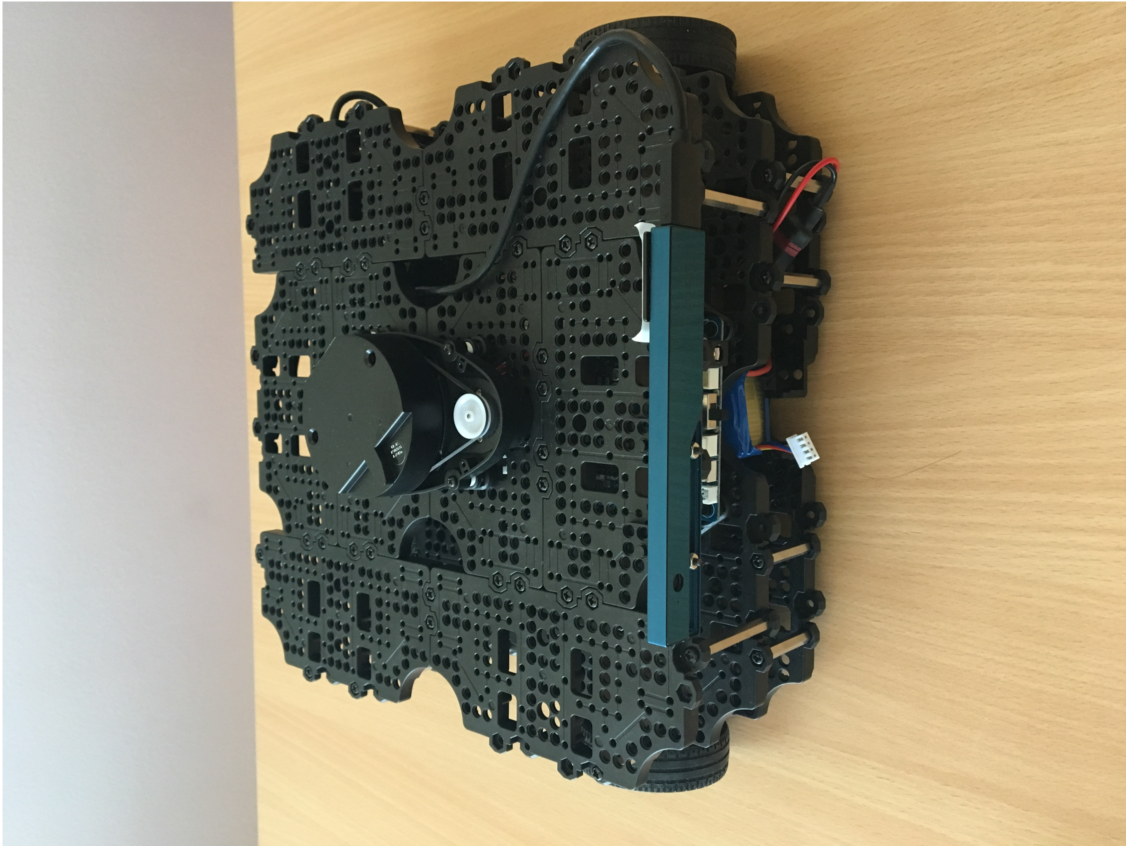


Figure 9.5: Turtlebot3 Reference robot.

9.3 Conclusion and future works

ROSploit is modeled after NMAP and Metasploit for modeling and exploiting vulnerabilities in ROS. This tool is designed to provide a framework for further research into security for ROS systems, allowing a flexible platform for future research development. This chapter demonstrated that the tool implements all of the previously discussed attack vectors. Additionally, we explore several of the implemented attack vectors concerning the CIA triad. For future work, RoSploit will be extended to support additional functionalities, newer attacks, as well as allow multi-stage exploits. Finally, we will extend the tool to support ROS2[**ROS2**].

9.4 Key Contributions to ROS-Immunity

ROSploit was combined with *ROSPenTO* [186] to address vulnerabilities (such as those listed in the RSF [128]) to codify the automatic vulnerability detection component of *ROS-Immunity*. Combining these tools provides users with a complete security scan, with *ROS-ploit* providing fingerprint scanner and *ROSPenTO* providing canned exploits.

Additionally, *ROSploit* was utilized to test the functionality of ROS-Defender and *ROS-FM*, to ensure that the tools functioned properly and defended against all known vulnerabilities as expected. *ROSploit* was the guiding tool for all defensive research conducted for *ROS-Immunity*, as it provided an easy way to locate security flaws.

Chapter 10

Discofuzzer

"Security is always excessive until it's not enough."

Robbie Sinclair

10.1 Introduction

As discussed in the previous chapter, it is vital to consider the safety-critical nature of robotic systems and how their robustness affects their physical environment. While vulnerability scanning for known vulnerabilities is the minimum, scanning for previously unknown vulnerabilities is essential to create actual security. The overarching goal is to prevent the exploitation of '0-day' vulnerabilities, which are presently unknown.

One such method of testing this robustness is through the use of fuzzing. Fuzzing is a core component, as it complements traditional unit testing. However, robotic systems have to face different threats that current fuzzers do not adequately address. Fuzzing has become a central tenant of computer security research over the past 20 years and consists in the repeated execution of the program using inputs sampled from the input space [147]. This repeated execution is performed automatically with the sampling of inputs adhering to different heuristics. The goal of fuzzing is to efficiently search for potential problems in software by testing responses to a wide variety of inputs while detecting problems in the outputs. In their most primitive form, fuzzers generate random data, pass it into the

program, and analyze if it continues to run. If the program continues to run, the fuzzer continues this by generating and applying another random set of data. If the program ceases running or crashes, the input that caused the crash is logged and reported, and the program is restarted. Fuzzing generally provides the user with the inputs that caused crashes for later analysis. While basic random input is the most primitive form a fuzzer can utilize, recent research has improved fuzzers by efficiently searching the input space and utilizing knowledge about the target program.

Concolic execution can help the exploration of the input space where standard approaches fail. These fuzzers are called hybrid and vastly showed their efficacy in testing software programs[63], [98], [130], [160]. However, the community also provided lighter methods to explore the program under test and overcome magic numbers and nested checksums. VUzzer [86] leverages control- and data-flow features to generate interesting inputs. TFuzz[119] solves the problem by removing sanity checks in the target program when it cannot bypass them. REDQUEEN [133] exploited the input-to-state correspondence, *i.e.*the feature of some program where parts of the input often end up directly in the program state. Some fuzzers try to use some level of knowledge of the input space and generate fewer but better inputs. TIFF [108] tags input bytes with its basic type (*e.g.*, 32-bit integer) in the program, then it uses this information to mutate the inputs accordingly. ProFuzzer [159] adds to this the ability to understand input fields of critical importance. Superion [158] creates meaningful tests for programs that process structured input (*e.g.*XML engines) by analyzing its grammar.

The differential testing approach is a way to find errors by comparing results from different implementations of the same program, resulting in cross-referencing oracles. Chen *et al.*[54] applied differential testing to a coverage-guided fuzzer for Java Virtual Machines. Nezha [83] exploits these differences also to generate inputs that are more likely to trigger a bug.

Sanitizers are another approach to detect specific failures in a fuzzing campaign. AddressSanitizer[23] finds out-of-bounds accesses to heap, stack, and global objects, as well as use-after-free bugs. UndefinedBehaviorSanitizer [167] detects null pointer, signed integer overflows, and wrong type conversions. MemorySanitizer [166] focuses on uninitialized

reads.

This chapter introduces a discontinuity-based fuzzer for ROS, *DiscoFuzzer*, a novel fuzzing methodology that exploits the physical world's continuity to automatically explore the input space and detect malfunctions in robotic software modules. The goal of *DiscoFuzzer* is to address external security threats that consist of an attacker gaining control of specific modules of a robotic system or interfering with the robot's sensors input data[72, 154, 152, 55]. An attacker can lead a robotic system to disregard safety measures, behave unsafely, or crash.

DiscoFuzzer directly addresses failure states, *i.e.*, unsafe behavior, that other fuzzers do not have the ability to address, and provides a mathematical approach to provide dynamic solutions. The fuzzer operates by performing a single test against a target and constructing a model of the targets' response behavior, including discontinuity analysis. Tests on several inputs can detect anomalies that are usually only unveiled by extensive manual testing.

At the time of this dissertation, *DiscoFuzzer* is the first fuzzer applied to ROS. However, Santos *et al.*proposed a property-based testing approach for ROS. Property-based testing is very similar to fuzz testing, but the latter does not require familiarity with the target to specify *properties* and *shapes*[125]. Currently, there are other fuzzers to address other components of the ROS middleware; however, most node-specific vulnerabilities are not visible to these fuzzers. *DiscoFuzzer* 's novelty is dynamically locating these node-specific vulnerabilities.

The novelty of *DiscoFuzzer* lies in taking the previous model-based testing approaches and extending their most commonly found failure modes into a highly search-based system.

10.2 Discontinuity-Based Fuzz Testing

Let us consider an industrial robotic arm with a single joint. The joint software module receives as input the desired angular position and effort (power) to provide to the joint motor to perform the movement. The output is an array describing the actual change of angular position, velocity, and acceleration in time as performed by the joint. Figure **??** shows an example of how the proposed methodology works. The discontinuity fuzzer begins by first choosing a range of continuous input values. In this case, the range consists of three values

for the angular position and three values for the effort. They constitute three fuzz test cases that are provided one by one to the joint software module of the robotic arm. Each defined test case wants to spin the joint one radian clockwise but with increasing effort, *i.e.* the power to provide to the motor to perform the movement. After each test, the output messages are collected, and their values are concatenated to create the output functions. The fuzzer then computes first and second derivative functions to analyze discontinuities and unveil anomalies in the software's behavior. Indeed, the output functions of velocity and acceleration contain a significant change of the slope. The fuzzer detects an anomaly that consists of the uncontrolled rapid rotation of the joint, which could lead to a burn out of the motor because the software did not check the physical constraint[1]. The methodology reports the properties of the output function to help guide input test-case generation.

This example illustrates the functionality of discontinuity fuzzing. Initially, this methodology identifies all of the target's inputs and outputs. Each input creates a sample generator to provide intervals, *i.e.* ranges of continuous values, depending on the input types (including arrays). Regarding floating points, the sample generator also needs to define the distance between values in the interval (*i.e.* the resolution). The methodology changes one input at a time while setting the others to default values. Thus, a fuzzer should perform the following actions:

- Choose an interval: The fuzzer samples a range of continuous values and generates this iteration's test cases (§10.2.1).
- Test values in the interval: The fuzzer performs one test at a time against the target and saves the outputs.
- Analyze the output distribution: The fuzzer analyzes all of the outputs of this iteration to detect anomalies (§10.2.2).

The fuzzer iterates on these actions until a termination criterion is met, such as a time limit.

[1] This is a simplification of bug #86 presented later in this chapter as part of the`universal-robot` benchmark in Table 10.1

In the following subsection, several approaches that were adapted to implement the above methodology are presented.

10.2.1 Sampling approaches

A sampling generator consists of an iterative process to choose the next group of samples to test. Three different sampling approaches were defined based on three numerical sampling functions known in the state-of-the-art literature.

First, a simple randomized Monte Carlo analysis[11] chooses every single value at random. Second, a Chebyshev approximation[9] is performed of the target's input/output function and use the discontinuity and root-finding of the Chebyshev approximation to look for potential areas of interest. Third, a model of the behavior function using a Cubic Hermite Spline Interpolation is constructed. These three functions were chosen to cover different spectra of potential erroneous behavior.

The sampling approach is different between arrays and individual values, and each is expanded separately below.

Monte Carlo Sample Generator

Monte Carlo is a method based on repeated random sampling. To conduct Monte Carlo sampling for individual values, the sample generator chooses a random value to be the central point for evaluation. Two values are sampled in either direction for derivation analysis, for a total of five values per iteration. For arrays, the fuzzer randomly chooses between (1) creating the array repeating a single sample point and (2) sampling a central point and creating the array with increments of the central point. Within this dissertation, Monte Carlo sampling is referred to as `monte_carlo`.

Chebyshev Sample Generator

Chebyshev is a numerical method that generates increasingly complex polynomials (Chebyshev polynomials) to interpolate a dataset. Since the initial introduction of the `chebfun` package in Matlab in 2005, Chebyshev polynomials have gained broad interest in the field of numerical approximation research as a way to model an arbitrary function. One of the vital functions that Chebyshev polynomials provide is the efficient location of roots and inflection

points, computed at each approximation. The Chebyshev sample generator initially returns a list of interpolation inputs. Once the target runs the interpolation inputs, the new list of interpolation inputs is generated based on the target outputs. Indeed, this approach recursively determines which points would be most optimal to sample, given the current function approximation. Within this dissertation, Chebyshev sampling is referred to as `chebfun`.

Spline Sample Generator

Similar to Chebyshev, Spline is a numerical method that interpolates the function through a piecewise polynomial called a spline. Spline sampling begins with the generation of two clusters of random values, in the same way as `monte_carlo` does. Then, it generates a cluster at the midpoint between these two. If the third selected cluster's output is similar to the spline interpolated output of the first two clusters, within a user-specified tolerance, then a spline is drawn between the first two clusters. Thus, it randomly generates a new cluster. Otherwise, it generates a new cluster within the bounds set by the first two generated clusters, and it performs a new interpolation. For arrays, it follows the same behavior as `monte_carlo`, with the caveat, the methods for filling arrays are treated as two distinct sources of input for interpolation and generation separately. Within this dissertation, Spline sampling is referred to as `spline`.

Structurally, the `monte_carlo` simulation has the lowest processing overhead and the fastest turnaround, allowing it to cover many potential inputs. However, the `chebfun` analysis provides a more precise analysis. The `spline` approach acts as a hybrid of the two, with moderate precision and processing overhead.

10.2.2 Discontinuity Analysis approach

Discontinuities in the output functions are perhaps the most significant source of issues as the difference in state course changes incredibly quickly as the system runs. The discontinuity analysis approach first looks for discontinuities in the output. It focuses on sharp changes in the output functions' slope: large values in the first derivative indicate a significant change, and large values in the second derivative indicate a significant effect. These changes in the derivatives are more likely to be indicative of errors than any other behaviors, as it is less

likely for a numerical output with a smaller average derivative change to be the source of vulnerability.

However, there could still be a chance that a minor change in derivative will create a significant effect. As cyber-physical systems tend to maintain a consistent internal state about the world which they use to make decisions, the approach also considers the amount of influence past inputs have on current outputs. This issue is more obvious to recognize with the input of large floating-point values: these values continue to affect the output regardless of the new provided value due to the mathematical rounding properties. Once an input corrupts the internal state, unexpected or malicious behavior becomes more likely in the long-term. Thus, the analysis raises an error when old inputs in the memory buffer are disproportionately different from new inputs.

Furthermore, anomalies where changes to one input cause changes in multiple outputs were of particular interest. The analysis was enhanced by considering how many times a field in the output topic has changed. If it changes at some consistent rate, then there is presumed to be a connection between the published topic and the output field. However, if there is a one-off outlier, it is more likely to be a memory or mathematical calculation error. For example, consider a memory-overflow vulnerability. If an input makes the software write beyond an array's bounds, it could overwrite the other inputs. Thus, a field could change the output response message when it would typically not be affected by that type of input.

In summary, this discontinuity analysis focuses on two main patterns: values changing very quickly and values changing unexpectedly.

In both experimental analysis[25] and industrial surveys[91], algorithmic bugs are found to make up between **30 and 50%** of the causes for robotic software failure. Additionally, previous exploratory research found that more then **75%** of all algorithmic bugs caused either a discontinuity in output ($>$50%) or a system crash($>$20%)[25]. The modular nature of most robotic systems means that while there may be discontinuities in a node's output, all downstream nodes need to adapt to these discontinuities. For example, a user may create a command to have a robot reset its position to account for rotation. In the real world, this may be advantageous to avoid errors after long runs. However, this leads to a discontinuity

in robot position, which may cause bugs to manifest in other robot components, potentially leading to robot burn-out or another failure. Even in these cases, where the user may see an immediate advantage, the importance of addressing an anomaly is vital in avoiding potential failures in other components.

10.3 DiscoFuzzer

DiscoFuzzer implements the presented methodology to target ROS nodes. The fuzzer is developed in Python 3.6 and targets ROS kinetic.

Figure 10.1 shows an overview of *DiscoFuzzer*. The fuzzer's input is a user-defined configuration file that includes target information. It outlines the type of messages used in the topics published or subscribed to by the target node. It defines the input topic for the sampling generators. The configuration file also contains how to execute the target node, *i.e.* flags, and arguments for the command-line. An orchestrator reads the configuration file, and it ① initializes the ROS environment with the target node and *DiscoFuzzer*'s publisher and subscriber. A timer starts to count how long the fuzzer should test the target. Then, the first iteration starts. The orchestrator ② notifies the test case generator of the new iteration. The generator samples a new interval, and it ③ provides one value at a time to the ROS environment through the publisher. Once the subscriber receives a message, it ④ copies this value into the output analyzer. The analyzer ⑤ sends the result of the analyses back to the orchestrator. When anomalies are detected, the orchestrator ⑥ relaunches the node and returns the node to a starting state so that testing can continue. The fuzzer's output is a set of reports issued during the fuzzing campaign.

10.3.1 Test Case Generator and Publisher

The test case generator takes the system's current state and generates a new set of inputs to test through the publisher. The user can define the *interval size*, *i.e.* the number of inputs to generate at every iteration. If the sampling generator computes the size algorithmically (*e.g.* in `chebfun`), it ignores this configuration parameter.

The generator tracks which inputs it has already given to the system to ensure that there are no unnecessarily repeated values. Beyond that, it maintains a collection of every single

combination of potential inputs from the topic to publish on. This collection allows for fuzzing of individual inputs one at a time and inputs in groups of size $s_2 ... s_n$, where *n* is the size of the input. It allows for identifying potential interactions or discontinuity in situations where the combined two inputs can break the system, even if they could not individually. It focuses on single inputs and implements a power-law drop-off for values fuzzed in combination, prioritizing potentially interesting interactions found from the single fields before conducting a random search. For every ten values fuzzed, nine values are single, and one is a combination. For the combinations, 90% are pairing, 10% are combinations greater than two.

The test generators implement the three sampling approaches presented in the methodology.

For the `monte_carlo` sampling approach, the fuzzer chooses the central input at random. The fuzzer stores every point, and it discards any new point that overlaps with the previous range. Additionally, the fuzzer was initialized with some common failure-inducing values such as `NaN`, infinity, and very large or very small floating points.

Since the `chebfun` approach works by calculating a polynomial and measuring the result, the generator relies on the information provided by the output analyzer (through the orchestrator). For this implementation, the `pychebfun` library[175] was chosen as it had the most active development and stars on GitHub outside of the Matlab core implementation. As the `chebfun` approach stores only the expansion coefficients, no space-saving optimizations are required for a longer run.

The `spline` approach consists of a numerical analysis that stores a group of splines containing both measured points and their derivatives, allowing for more accurate interpolation. One of this method's main benefits is that it allows arbitrary N-dimensional interpolation of functions without extra computation. The `spline` interpolation permits performing a more accurate analysis of array fields. For this implementation, the `scipy` interpolation library [176] is used, choosing points for the initial array in a random method, just like the `monte_carlo` approach. The `spline` interpolation approach is refined with a mixture of midpoints and random additional points.

For generating an interval of floating points, *DiscoFuzzer* requires the user to choose a

resolution. The resolution represents the distances between points in the same interval and indicates the number of decimal digits to consider.

The publishing rate is limited in two areas: (1) the rate at which the targeted node can respond and (2) the rate at which the subscriber can update. While it is possible to optimize the subscriber and assign new threads to it, nothing can be done about the targeted node. As such, the publisher runs on a single thread, and no additional processing capabilities are required.

Users can also choose to set limitations on the `monte_carlo` sampling if they expect that only specific values will produce exciting results. Users can set a *compression factor* and/or place *constraints on combinations*.

In the first case, *DiscoFuzzer* will choose a range at random with a size equal to the compression factor with bounds determined by the *resolution*. Then, *DiscoFuzzer* sweeps through the range from bottom to top at the chosen resolution, only saving the center of the range instead of saving each selected random point. Effectively, whenever *DiscoFuzzer* chooses a value for its sampling, it chooses the next value in the specified range instead of a completely random value. Once the range has been fully explored, it will randomly select another range and repeat the sweeping process. This gives users far more space-efficient storage for their simulation at the cost of far fewer random values being explored over time. As an example, suppose the user sets a range size of 100. When using the *compression factor* feature, after sampling 10,000 points, it would only have 100 values saved in memory instead of the 10,000 total values it would have under normal sampling procedures.

The *constraints on combinations* feature allows users to set constraints on generated inputs. These can vary in complexity from simple constraints such as limiting fields to a specific range of values (such as $x < N$) to specific combination rules (such as $y > x; y < N$). Users can specify any numeric constraints they deem necessary for their purposes. This allows users to better guide fuzzing processes to be more efficient.

10.3.2 Output analyzer and Subscriber

The output analyzer takes the outputs captured by the subscriber and analyzes them to provide reports to the orchestrator.

The analyzer operates by creating a mapping of potential discontinuities and their inputs. It begins by establishing a baseline for every single output topic, by looking for integer, floating-point, Boolean, and array outputs, and stores a shortened differential vector for them. First, it collects the five (for `monte_carlo` and `spline` sampling approaches) or N (for `chebfun` sampling approach) values. It starts the anomaly detection process in a separate thread while signaling to the orchestrator to send a new group of values. This ensures that the system is utilizing as much of the duty cycle as possible.

The anomaly detection algorithm is at the center of this concept. Reliance is placed primarily on numerical approximation rules and analysis of previous research on bugs behavior in cyber-physical systems. The criterium consists of (1) numerical discontinuities *i.e.* points where the difference between two values is significant and nonlinear, and (2) information leakage across values in a message, *i.e.* when a field that is previously unchanged suddenly changes after input changes, especially when it is an outlier or a unique instance.

The discontinuity measures come from the underlying assumption of the continuous nature of the physical world. While there are valid reasons for discontinuity in a robot's output, large values are observed in the first or second derivative; mainly when such values are not reflected in their neighboring value derivatives, it necessitates further investigation. These discontinuities are often a sign of a bug in the control loop or floating-point mathematical computations. The quintessential example of this is a division that is rapidly approaching infinity as its denominator tends toward zero. The first derivative was used to look for any logical jumps, and the second derivative to look for any potential significant changes in the first derivative. The analyzer records the mean and the standard deviation for the first and second derivatives of each output. Any deviations more than a *detection threshold* are raised as a potential anomaly.

Furthermore, the analyzer includes a Bayesian sensitivity analysis[5] that calculates the weighted effect of each of the past N inputs on the output. The weighted effect is calculated

by measuring the output's derivative effects from a given group of inputs. This weighted effect is stored in an array of size N, and the sum of this array is always one. The weight array is continually updating, maintaining size N. Empirically, the first 50% of values were considered "old", and the second 50% of values were considered "new". The analyzer raises an anomaly if the sum of the "new" values is less than the *weighting threshold*. This indicates the process is no longer updating that output for new inputs, flagging a potential anomaly.

DiscoFuzzer considers `NaN` and infinity outputs as leading special cases of discontinuity. These values are vital for analysis, given that once an internal robotic state system starts to propagate `NaN` values, it is highly likely that it will continuously output `NaN` values. Unless the system has proper handling functionality, the same applies to values of infinity. The tool only flags `NaN` and infinity values when they occur in more than three consecutive listings, as this was determined to be a sign that the robot is unable to perform any new calculations.

The analyzers also look for anomalies where changes occur unexpectedly in some outputs. Upon startup, a mapping is created for every input and output field combination. This mapping is one-to-one or one-to-many. When a new message is received from the subscriber, a counter is incremented for those particular mappings for every field changed. The analyzer raises an anomaly when the counter is incremented, and the counter value is below a *normality threshold*, a user-defined ratio of the maximum difference between any two mappings for the same input field. A low counter indicates that a value has changed when it remains typically unchanged, a potential anomaly.

By default, all three thresholds (*detection*, *weighting*, and *normality*) are set to be equal to two σ or two standard deviations from the mean. This value is based on the idea of using a 95% confidence interval for fuzzing anomaly detection from Zhao *et al.*[160]. The user can customize each threshold. For example, a user can specify the *detection threshold* and *weighting threshold* at two σ, but set the *normality threshold* to three σ, if an output field has a low rate of change causing a larger number false positives than expected.

Once the anomaly detection algorithm has discovered an anomaly, the analyzer saves inputs, outputs, and timings of the test, as a pickled message object.

10.3.3 Crash Detection

DiscoFuzzer implements another anomaly detection mechanism, performed by the orchestrator, that fuzzers use typically: crash detection. The orchestrator deploys the ROS nodes in python subprocesses. When a node crashes, the subprocess returns with an error code that is caught by the orchestrator. Furthermore, the orchestrator periodically uses the utility *ROStopic*[174] to check whether the topics are still available to the nodes. When a crash is detected, it creates similar reports every time it discovers an anomaly.

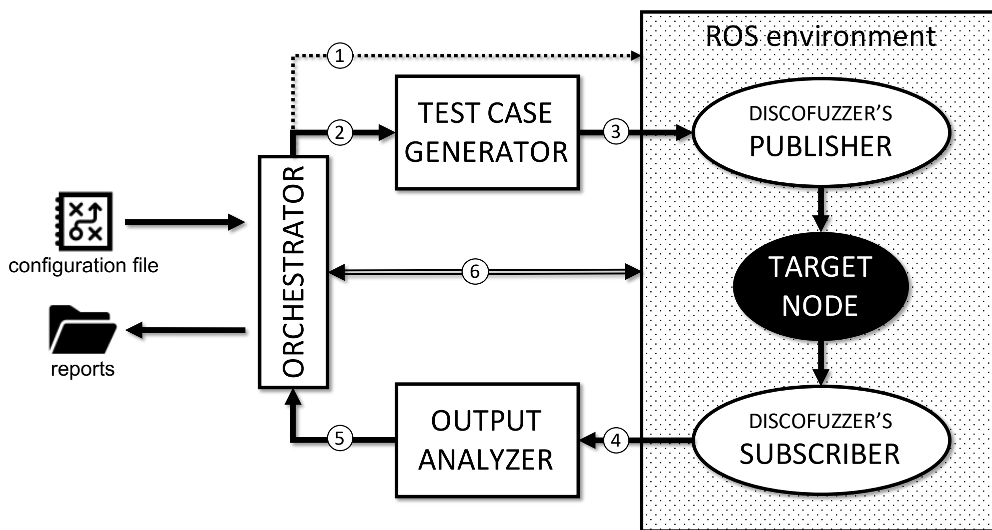


Figure 10.1: Overview of DiscoFuzzer

10.4 Evaluation

In this section, *DiscoFuzzer* is evaluated for the effectiveness of the discontinuity-based analysis approach and the efficiency of the three sampling approaches.

More specifically, an experimental campaign was conducted to answer two main research questions:

RQ1 Can *DiscoFuzzer* detect vulnerabilities in ROS nodes based on discontinuity analysis? (§10.4.3)

- **RQ1.1** How many bugs does the discontinuity analysis detect compared to the other analyses, previous work and human detection?

RQ2 Which sampling approach used by *DiscoFuzzer* is more efficient to trigger vulnerabilities in ROS nodes? (§10.4.4)

A set of benchmarks was created by including popular ROS nodes with known vulnerabilities (§10.4.1). Then, *DiscoFuzzer* was executed against the benchmarks (§10.4.2). Furthermore, some common use-cases and threats to validity are considered to conclude the evaluation(§10.4.5,§10.4.6).

10.4.1 Benchmarks

Table 10.1 shows the benchmarks.

Fourteen (14) open-source ROS packages in the ROS ecosystem were identified, using the rosmap[150] tool. Ten (10) of them were chosen by determining the packages that had the highest number of packages that depended on them, *i.e.*those packages with a very high impact on the community. These ten core packages are usually included in academic robotic systems to support research, development, and prototyping[28]. Additionally, four (4) additional real-world packages were chosen that are used in non-academic contexts. These packages are highly-rated on GitHub, including the Autoware self-driving car, the Udacy self-driving car, the NASA Mars rover, and the ARDrone flight system.

All code for the 14 packages is hosted and regularly updated on GitHub. For every package, its issue tracker was checked for closed issues. Only those issues related to

Table 10.1: *DiscoFuzzer*'s benchmarks. The first two columns enumerate the names of the benchmarks. The third and fourth columns show the URLs of the code and the lists of the ids of each benchmark's vulnerabilities. An exception occurs for the benchmarks number 5 (`carla`) and 6 (`carla-bis`): they require other code running to be tested, namely `carla-core`, introducing in the above-mentioned benchmarks two new vulnerabilities. We added the URL of `carla-core` in the table. We also added the previously unknown vulnerabilities in bold (*cfr.* §10.4.3).

	name	URL
1	apm-planner	`https://github.com/ArduPilot/apm_planner/tree/dfe1865a82`
2	ardupilot	`https://github.com/AutonomyLab/ardrone_autonomy/tree/2e3b75a`
3	autoware	`https://github.com/autowarefoundation/autoware/tree/31f4bfb`
4	autoware-bis	`https://github.com/autowarefoundation/autoware/tree/e625625`
5	carla	`https://github.com/carla-simulator/ros-bridge/tree/8e468ca`
6	carla-bis	`https://github.com/carla-simulator/ros-bridge/tree/625960e`
	carla-core	`https://github.com/carla-simulator/carla/tree/ec3bb90`
7	cartographer-ros	`https://github.com/googlecartographer/cartographer_ros/tree/2538ac`
8	cartographer-ros-bis	`https://github.com/googlecartographer/cartographer_ros/tree/7bcdda`
9	cob-driver	`https://github.com/ipa320/cob_driver/tree/7a5d7c8`
10	image-pipeline	`https://github.com/ros-perception/image_pipeline/tree/d11edf3`
11	lsd-slam	`https://github.com/tum-vision/lsd_slam/tree/bb82258`
12	moveit	`https://github.com/ros-planning/moveit/tree/ece11fe`
13	mrpt	`https://github.com/mrpt/mrpt/tree/f564006`
14	mrpt-bis	`https://github.com/mrpt/mrpt/tree/a4bcb08`
15	mrpt-tris	`https://github.com/mrpt/mrpt/tree/31e853f`
16	navigation	`https://github.com/ros-planning/navigation/tree/48323b0`
17	open-source-rover	`https://github.com/nasa-jpl/osr-rover-code/tree/33f072e`
18	rtabmap	`https://github.com/introlab/rtabmap/tree/173bd49`
19	rtabmap-bis	`https://github.com/introlab/rtabmap/tree/344dc16`
20	universal-robot	`https://github.com/ros-industrial/universal_robot/tree/8c912d4`

software vulnerabilities were retained, and all others were discarded (*i.e.*user's questions, feature requests, or compilation issues). Each vulnerability was assigned a numeric and unique id. A benchmark was defined as the code in the package's repository at a version that includes the maximum number of vulnerabilities not already included in other benchmarks. This criterion resulted in more benchmarks for the same package but at different versions. Thus, 20 benchmarks were identified, with 89 vulnerabilities.

During the execution of the fuzzing campaign, further vulnerabilities were detected in the targets. For each of them, the issue trackers were searched to find the related issue. If not found, the new vulnerability was considered to be previously unknown to the community and submitted as a new issue. They were assigned a numeric and unique id and added to *DiscoFuzzer* 's benchmarks. Since the 8 vulnerabilities were reported to the developers, 5 have been officially confirmed as potential security threats. Of those 5, 3 have been patched, and the other 2 are still open issues. The other 3 are still pending confirmation.

10.4.2 Experimental design

discofuzzer's configuration parameters were tuned with results from all three different sampling approaches. Table 10.2 shows the values of all other parameters that are identical among the three. These parameters were determined experimentally after exploratory analysis with *DiscoFuzzer*.

For each combination of a benchmark and one of the three *DiscoFuzzer* sampling ap-

Table 10.2: *DiscoFuzzer*'s configuration parameters used in the evaluation.

parameter	value
interval size	5
resolution	0.01
compression factor	200
detection threshold	2 times the standard deviation of the expected values
weighting threshold	3 times the standard deviation of the expected values
normality threshold	2 times the standard deviation of the expected values

proaches, the fuzzer was run ten (10) times. A single repetition, *i.e.* a fuzz campaign[147], lasts 24 hours. The only exception is the `chebfun` sampling approach: it only needs to run once per benchmark because of its deterministic nature. In total, 420 fuzz campaigns were run, 420 full days in CPU time.

The fuzzing environment runs in a virtual machine with four cores of an Intel(R) Xeon(R) CPU E5-4650 set in a host emulation, 8 GB of RAM, Ubuntu 16.04, ROS kinetic, Python 3.6, and pychebfun [2]. The program's execution was optimized with the python profile library and implement an automatic load balancer between the subscriber and the publisher to ensure that the system is continuously publishing at the highest rate that the fuzzed node can support. The testing time was enforced by using the system clock to terminate the program after 24 hours.

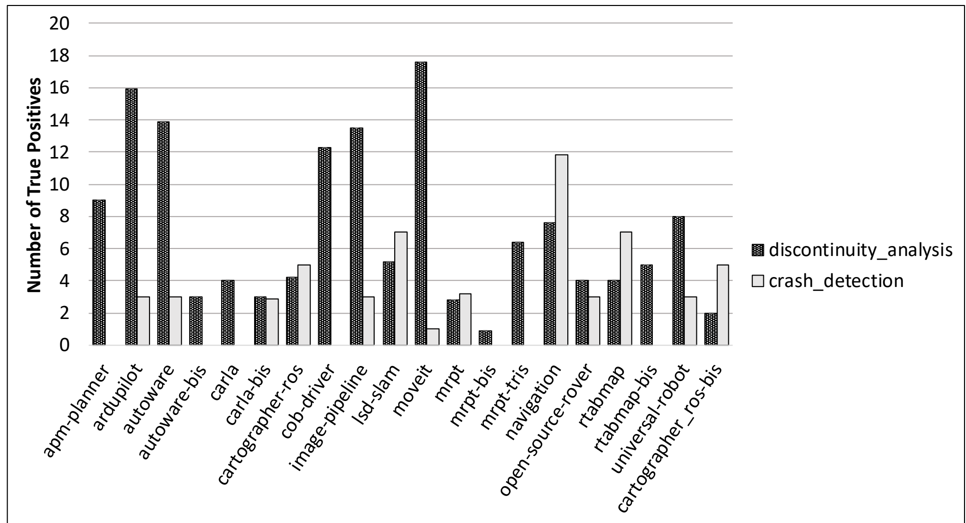
10.4.3 Effectiveness of DiscoFuzzer

Every fuzz campaign produced a report for each vulnerability detected by *DiscoFuzzer*. The report contains information such as sampling approach, target node, last messages (i/o), anomaly type, and detection time (elapsed time since the beginning of the fuzzing campaign). Each report was inspected manually to test its reproducibility. The report was labeled with its vulnerability-id if it produced a True Positive. In case of a previously unknown vulnerability, a new one was created and recorded in the benchmarks (*cfr.* §10.4.1.

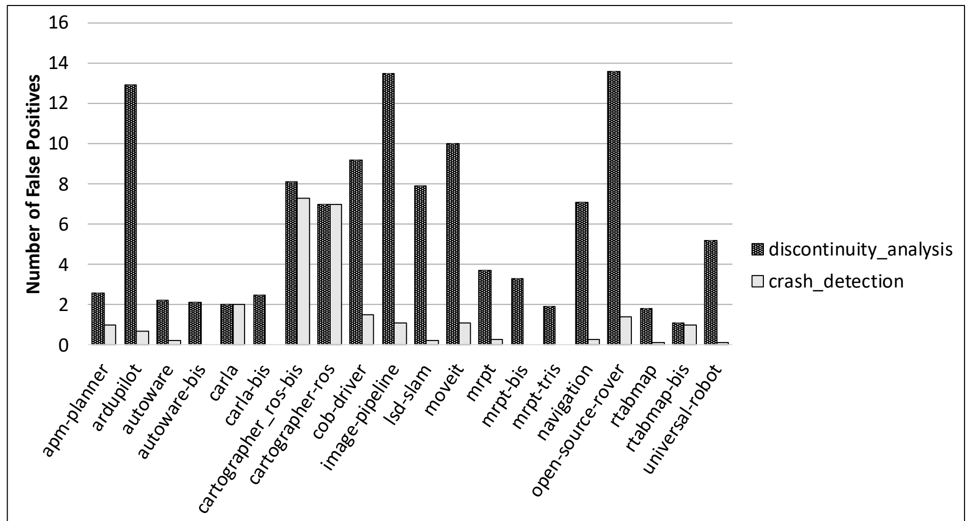
DiscoFuzzer found eight previously unknown vulnerabilities, marked in bold in Table 10.1, and identified 77 of the 89 previously known vulnerabilities. The reports were grouped by the issuing detector, *i.e.* either `discontinuity_analysis` or `crash_detection`. The crash detection mechanisms detected 22 distinct vulnerabilities: 20 of the 89 previously known vulnerabilities and 2 new ones. During the evaluation, the discontinuity analysis of *DiscoFuzzer* detected 63 distinct vulnerabilities: 57 of the 89 previously known vulnerabilities, and 6 new ones (*cfr.* **RQ1**).

Figure 10.2 shows the number of true and false positives, and the precision computed as the number of true positive divided by the number of all positives. Even if the `discontinuity_analysis` can detect more vulnerabilities in 12 out of 14 benchmarks, its complexity and infancy are

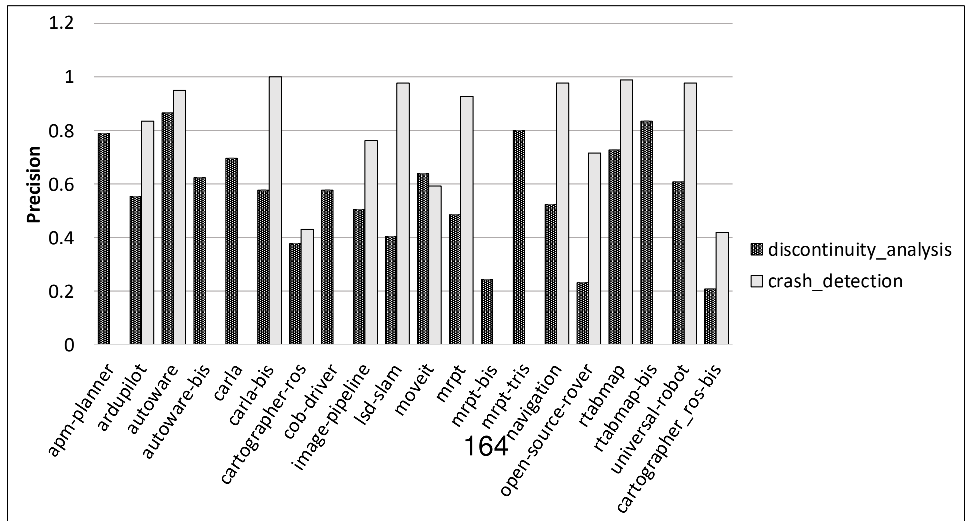
[2] The used version of pychebfun is at `https://github.com/pychebfun/pychebfun/commit/cda9283`.



(1) Number of True Positives



(2) Number of False Positives



(3) *DiscoFuzzer*'s Precision

Figure 10.2: Metrics grouped by the issuing detector mechanism. The bars' height represents the mean value for the group in the specified benchmark.

Table 10.3: Results of the statistical analyses on precision performed among *DiscoFuzzer*'s three sampling approaches, namely `spline` (s), `monte_carlo` (mc), and `chebfun`(c). The p is the p-value of the Mann-Whitney U test performed, while the A is the Vargha and Delaney's statistic. The first column lists the benchmarks' names.

Benchmark	$p_{s,mc}$	$A_{s,mc}$	$p_{s,c}$	$A_{s,c}$	$p_{mc,c}$	$A_{mc,c}$
apm-planner	0.106	0.34	0.481	0.50	**0.045**	**0.70**
ardupilot	0.380	0.55	0.115	0.65	**0.006**	**0.80**
autoware	0.435	0.47	**0.027**	**0.75**	**0.016**	**0.78**
autoware-bis	0.432	0.47	0.099	0.66	**0.049**	**0.70**
carla	0.111	0.65	0.121	0.65	0.227	0.40
carla-bis	0.221	0.41	0.226	0.60	**0.032**	**0.73**
cartographer-ros	**0.036**	**0.26**	0.323	0.43	0.000	0.94
cartographer-ros-bis	0.154	0.36	**0.000**	**1.00**	**0.000**	**1.00**
cob-driver	0.380	0.46	**0.000**	**1.00**	**0.000**	**1.00**
image-pipeline	0.470	0.48	**0.000**	**1.00**	**0.000**	**1.00**
lsd-slam	0.351	0.56	**0.000**	**1.00**	**0.000**	**1.00**
moveit	0.296	0.42	**0.057**	**0.30**	**0.035**	**0.30**
mrpt	0.468	0.52	0.180	0.38	0.148	0.36
mrpt-bis	**0.000**	**0.94**	-	-	**0.000**	**0.94**
mrpt-tris	0.251	0.42	0.339	0.56	**0.084**	**0.67**
navigation	0.114	0.34	0.221	0.60	**0.001**	**0.90**
open-source-rover	0.350	0.45	0.220	0.40	**0.000**	**0.00**
rtabmap	**0.073**	**0.33**	0.288	0.42	0.228	0.59
rtabmap-bis	0.448	0.48	0.500	0.49	0.482	0.51
universal-robot	0.409	0.54	0.221	0.60	0.219	0.40

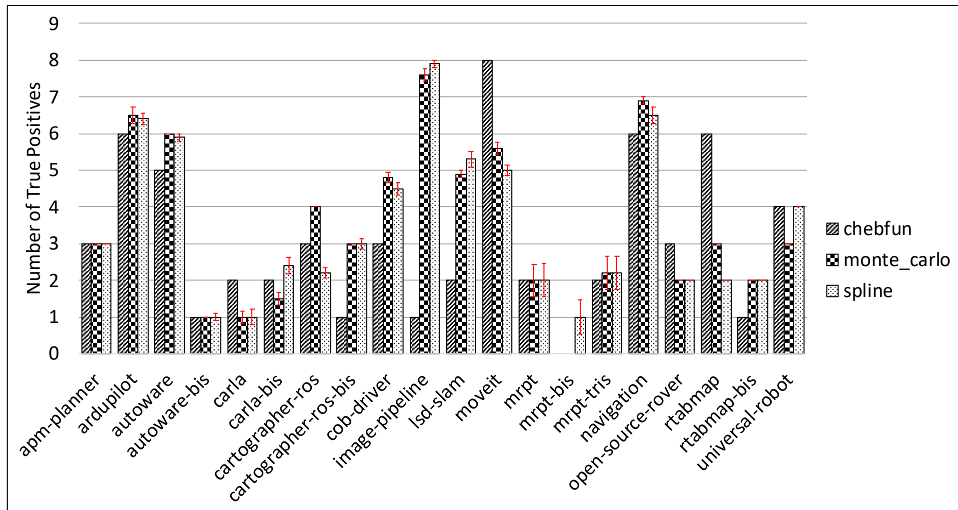
highlighted by its precision. The `crash_detection` almost has no false positives resulting in a much higher precision than `discontinuity_analysis`.

The discontinuity analysis of *DiscoFuzzer* detected 41 more unique vulnerabilities compared to the crash detection (*cfr.* **RQ1.1**).

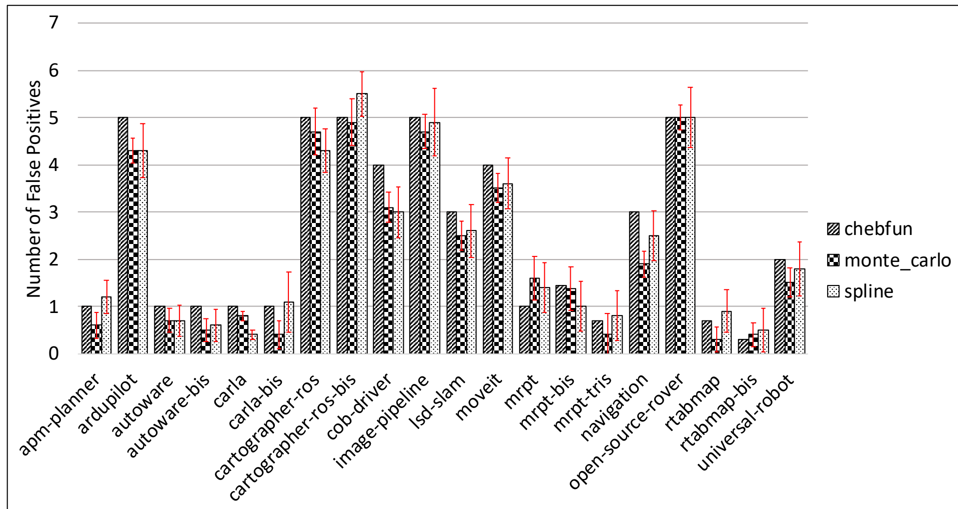
10.4.4 Efficiency of the sampling approaches

The reports were grouped by the used sampling approaches, shown in Figure 10.3.

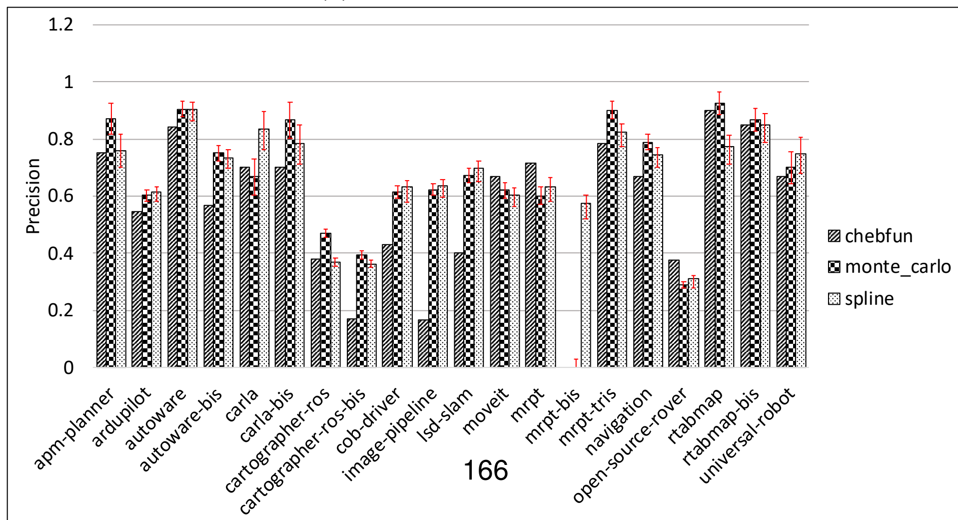
Following the methodology presented by Klees *et al.*[113], statistical analysis was performed to determine which of the sampling approaches has better precision. The Mann Whitney U test [1] was used to test the null hypothesis H_0 that the precision of two sampling approaches is statistically equal. In case of $p_value < 0.05$, H_0 was rejected. Vargha



(1) Number of True Positives



(2) Number of False Positives



(3) *DiscoFuzzer*'s Precision

Figure 10.3: Metrics grouped by the used sampling approach. The bars' height represents the mean value for the group in the specified benchmark. They also include an error bar, but for `chebfun`, that is deterministic.

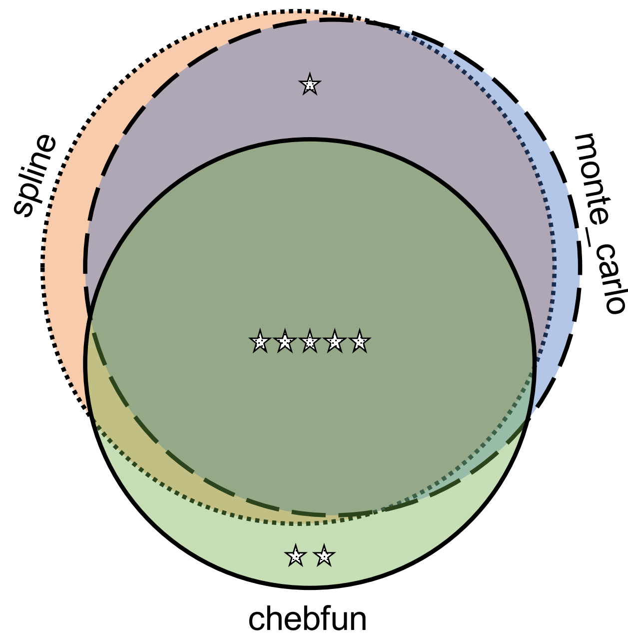


Figure 10.4: Venn diagram of the vulnerabilities as detected by the three sampling approaches. The area occupied by the three circles represents the number of unique vulnerabilities found. Overlapped areas indicate the number of unique vulnerabilities found by both (or the three) approaches. The stars represent unique vulnerabilities that were previously unknown.

and Delaney's A statistic [8] was used to compute the statistically different group's effect size. Given two groups, $A_{1,2}$ is close to 1 if the first group has statistically higher precision than the second group, close to 0 in the opposite case. Table 10.3 shows the results of this statistical analysis. The precision of the `spline` and `monte_carlo` sampling methods are statistically different only in three cases, where `spline` is more precise during a single benchmark and `monte_carlo` in other two. In 5 out of 20 benchmarks, the `spline` approach has statistically higher precision than `chebfun`, but in one benchmark, the converse is true. In 12 out of 20 benchmarks, `monte_carlo` is better than `chebfun`, but in the other two cases, the contrary holds. The `monte_carlo` sampling is the most precise, while using `chebfun` leads to the least precise reports. However, in the majority of cases, the statistical tests are inconclusive.

Table 10.4 and Figure 10.4 present the vulnerabilities as detected by the three sampling

approaches. Every sampling approach found unique vulnerabilities that the other two could not. In particular, the `chebfun` sampling approach found more than double the amount of these unique vulnerabilities, including two previously unknown vulnerabilities. In total, eight new vulnerabilities were detected, five of which were detected by all three methods, two which were identified by `spline` and `monte_carlo`, and two that were identified only by `chebfun`.

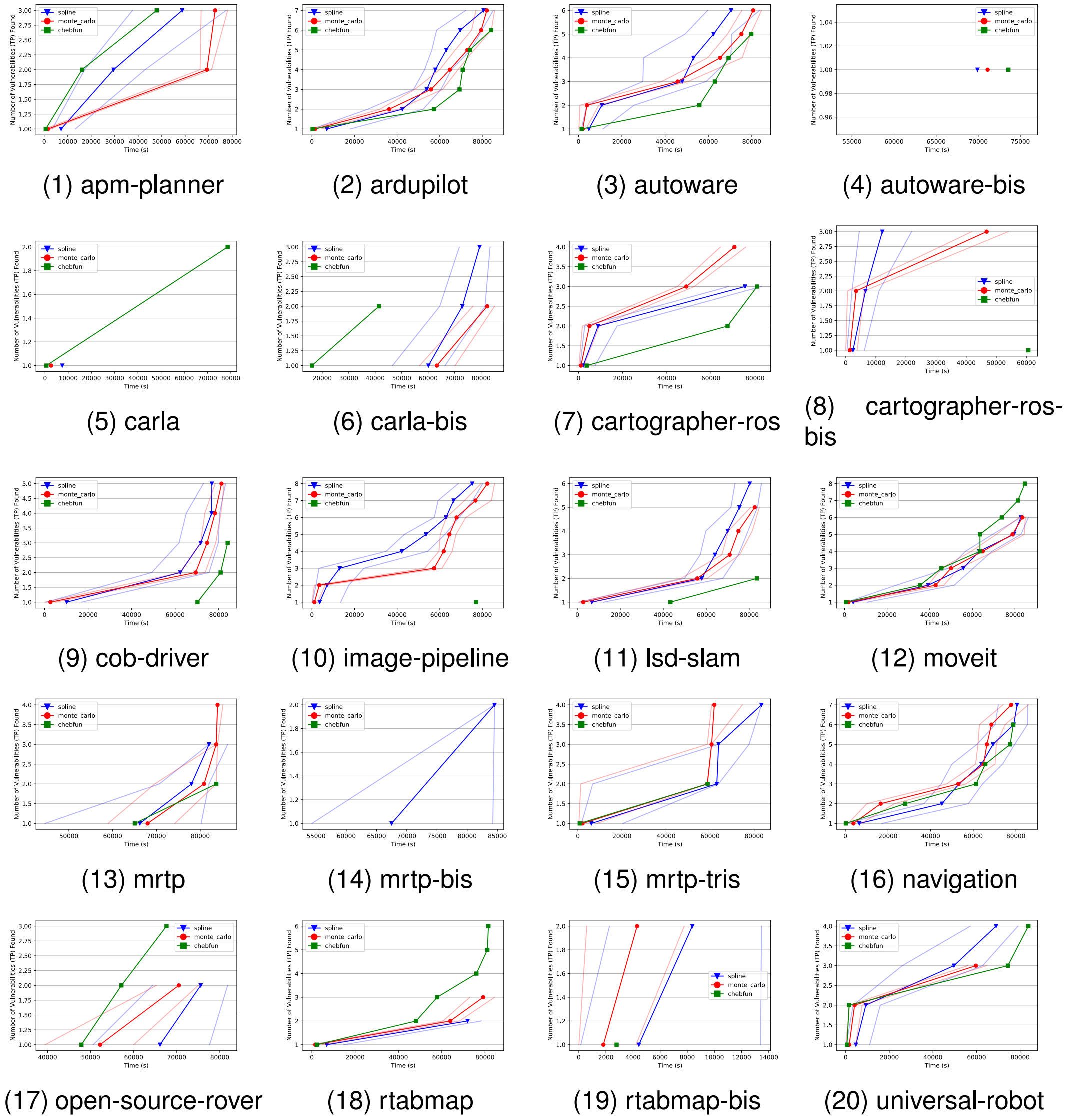
Similar to the precision analysis, the three sampling approaches' vulnerability detection times were analyzed through statistical tests. The null hypothesis H_0, for the Mann-Whitney U test, is that the detection times for a vulnerability in two sampling methods are statistically equal. The Vargha and Delaney's $A_{1,2}$ statistic is high if the first group has a longer detection time than the other, low in the opposite case. Then, each sampling approach was compared with the other two. For each comparison, the number of times it is statistically faster than the other was computed. Results of the statistical analysis of vulnerability detection times are in the appendix. Table 10.5 lists the results of these comparisons, demonstrating that `chebfun` approach is statistically faster to detect bugs. Figure 10.5 provides a view on the detection times for the benchmarks.

No sampling method is statistically better in finding vulnerabilities, but each discovered vulnerabilities the others could not. However, the `chebfun` approach occurs to be the fastest in finding vulnerabilities (*cfr.* **RQ2**).

10.4.5 Discussion

DiscoFuzzer demonstrates its capabilities in detecting previously known and unknown erroneous behavior. The three sampling approaches complement each other well to identify many types of cyber-physical vulnerabilities. In the majority of cases, *DiscoFuzzer* successfully identified vulnerabilities with high precision. However, testing revealed some case-specific behaviors that are notable for future analysis.

When considering the security of ROS systems, there are three viewpoints to analyze. First of all, ROS systems are insecure and provide little protection against attackers [72]. The second point of view is that through leveraging all security modules available through



(1) apm-planner (2) ardupilot (3) autoware (4) autoware-bis

(5) carla (6) carla-bis (7) cartographer-ros (8) cartographer-ros-bis

(9) cob-driver (10) image-pipeline (11) lsd-slam (12) moveit

(13) mrtp (14) mrtp-bis (15) mrtp-tris (16) navigation

(17) open-source-rover (18) rtabmap (19) rtabmap-bis (20) universal-robot

Figure 10.5: Detection times plotted for each benchmark (ROS node) and grouped by sampling approaches. The lighter lines around `spline` and `monte_carlo` represent the minimum and the maximum functions, while the darker lines are the average functions.

ROS [64], it was valid to treat these systems as reasonably secure. The only concern is sensor-based attacks. As an extension of the second viewpoint, the third one only considers attacks that could potentially leave long-lasting or hard to repair damage to the physical systems or surroundings. *DiscoFuzzer* operates under the assumption that the only vul-

nerabilities in ROS systems concern sensor-based attacks. Thus, every anomaly found by *DiscoFuzzer* is not adequately protected by the current security measures and can be exploited to cause damage to the robotic system and its surroundings. In this way, *DiscoFuzzer* provides another layer of protection.

Overall, false positives are exceedingly rare for crash failures. Two scenarios cause them: (1) virtual memory overflow due to dynamic array sizes, and (2) asserts. Most nodes only crash due to the first scenario occurring when the fuzzer passes a message with an extensive array causing the virtual machine to run out of memory (as they do not have swap). Then, the fuzzer crashes but does not indicate a bug in the node code. Notably, Google Cartographer exhibits a different crash profile. Google Cartographer uses asserts while parsing messages. For any malformed message, the assert causes the node to cease execution. After a manual analysis of the node's design and code, it was hypothesized that this is a design decision made to enforce well-formatted inputs. Even applying constraints and automatically eliminating any combination that caused too many crashes, the tool still picks up a moderate amount of false positives.

Regarding false positives for discontinuity analysis, they occur primarily as a product of the threshold settings, with some behaviors producing more noise than others. However, it is necessary to note that for *DiscoFuzzer*, a consistent threshold was chosen for testing of all packages. In real-world applications, users can tune the fuzzer for their specific applications.

In general, false negatives are an expected component of any testing platform[113]. Here, false-negatives were found to be primarily due to specific configuration requirements or bugs that required multiple fields to be at precise values.

A limitation of *DiscoFuzzer* is data-type selection. *DiscoFuzzer* only analyzes numeric data types, such as integers, floating points, boolean values, and arrays. Other data types are, therefore, excluded. While this approach behaves well for the stated goals, it deliberately excludes vital data types such as strings, a typical application in fuzzing research. A future version of the tool can mitigate this limitation by using a combination approach with other well-established non-numerical fuzzers such as AFL[51]. Another limitation concerns *DiscoFuzzer* 's strict adherence to the ROS message format. These tests never included

malformed messages nor out-of-order messages. Only valid messages were sent to the node to ensure that the fuzzer was only analyzing the node-specific code and not the ROS parsing mechanisms. These concerns are ROS specific and deemed out-of-scope as the goal was to create a generalizable technique.

The final limitation is due to the black-box nature of the test system. While black-box testing allows for easy integration of a wide variety of ROS nodes written in multiple programming languages based purely on input-output behavior, an extensive corpus of literature focuses on grey-box fuzzing, which is not explored here. Grey-box fuzzing would allow for more efficient guided fuzzing at the cost of complexity of fuzzer design and portability. While the use of grey-box fuzzing, in this case, would allow more accurate tuning of parameters, there would be a high cost in efficiency and performance. Using the black-box technique, accurate results were obtained while providing an efficient and usable tool for users.

While previous work on property-based fuzzing is valid, *DiscoFuzzer* outperforms them in detecting vulnerabilities that combine multiple properties in unison, which traditional property-based fuzzing detects less efficiently [125]. Additionally, *DiscoFuzzer* is more accessible for developers to validate discovered vulnerabilities. Similarly, another recent advancement in the field includes derivative-based fuzzing [146]. However, this approach is narrow in scope, focusing only on the control loop of systems. *DiscoFuzzer* is capable of a broader scope in its analysis, able to identify a more significant number of potential vulnerabilities with applications among any ROS package.

10.4.6 Threat to validity

The only threat to validity for *DiscoFuzzer* is external. As the initial design's focus was on ROS, it was not confirmed that these discontinuity results are generalizable to other cyber-physical systems. However, it is believed that the approach is general enough to apply to any system with well-defined inputs and outputs interfaces.

Table 10.4: Vulnerabilities as detected by the three sampling approaches. The first three columns indicate whether the row is the intersection (●) or not (○) of the elements detected by `spline`, `monte_carlo`, and `chebfun`. The fourth column has the number of vulnerabilities in the resulted subset. The last column contains the ids of the vulnerabilities, highlighting in bold the previously unknown vulnerabilities.

`spline`	`monte_carlo`	`chebfun`	#	vulnerability IDs
●	●	●	46	1, 2, 7, 8, 9, 10, 12, 13, 15, 16, 17, 21, 23, 26, 27, 34, 35, 39, 40, 53, 54, 55, 56, 57, 59, 60, 61, 62, 64, 66, 67, 68, 72, 73, 74, 81, 82, 86, 88, 89, **90**, **92**, **93**, **96**, **97**
●	●	○	21	11, 14, 20, 25, 29, 32, 33, 36, 37, 41, 42, 43, 44, 45, 47, 49, 50, 52, 70, 83, **91**
●	○	●	4	18, 24, 71, 85
○	●	●	3	19, 28, 80
●	○	○	3	48, 52, 63
○	●	○	2	46, 65
○	○	●	7	22, 30, 75, 77, 79, **94**, **95**

Table 10.5: Pairwise comparison among the three sampling approaches of *DiscoFuzzer*. While the second column present the results based on statistical significance, the last column presents the number of ties. There is a tie in a comparison of vulnerability detection times if such a comparison is not statistically significant, *i.e.*the p-value of the associated Mann-Whitney U test is greater than 0.05.

Comparison	Result	Ties
`spline` - `monte_carlo`	15 - 25	26
`spline` - `chebfun`	11 - 19	19
`monte_carlo` - `chebfun`	13 - 17	18

10.5 Conclusion

In this chapter, *DiscoFuzzer* was introduced in detail. *DiscoFuzzer* uses three different sampling methods and implemented targeting ROS packages. Here, it was tested against 14 ROS packages to evaluate its efficacy and efficiency. Results show that *DiscoFuzzer* can find more vulnerabilities than traditional crash detection mechanisms without the necessity to specify any ad-hoc property for the targets. However, no sampling approach resulted entirely better than the other, but all of them are necessary to find the majority of unique vulnerabilities. Thus, *DiscoFuzzer* will be able to use a combined approach for sampling and cover all the detected vulnerabilities at once. Furthermore, future research should improve discontinuity-based analysis and decrease the number of false positives while using the approach.

10.6 Key Contributions to ROS-Immunity

DiscoFuzzer is used as a supplement to well-utilized fuzzers, as it addresses an area of vulnerabilities that other fuzzers are incapable of identifying. The unique nature of ROS nodes makes the use of *DiscoFuzzer* vital in vulnerability detection. Finding new vulnerabilities requires many different tools working together. *ROS-Immunity* utilizes *DiscoFuzzer* along with the industry's best tools to provide an all-encompassing novel vulnerability search.

Similarly to *ROSploit* , the tool is setup for automatic scriptable runs so that users can just 'fire and forget.' Users do not have to continually monitor *DiscoFuzzer* during use, providing an opportunity for vulnerability detection in the background.

The novel vulnerabilities discovered by *DiscoFuzzer* were added to the database of novel vulnerabilities for *ROS-Immunity*., extending the number of vulnerabilities that *ROS-Immunity* was able to detect and mitigate.

Chapter 11

Discussion and Conclusions

"This was it. Finally, everyone was working together"

John C. McCrae

In this dissertation, *ROS-Immunity* and the designs of each of its respective components were explored and discussed in depth. This dissertation focused primarily on ROS vulnerabilities, how to detect, mitigate, and secure systems from them. Together, all of the chapters in this dissertation describe the development of *ROS-Immunity*, an integrated tool for securing and monitoring any arbitrary ROS system.

Figure 11.1 outlines the development pathway that led to *ROS-Immunity*. All components concisely contributed to this pathway. Many tools, such as *ROSploit* influenced the development of later tools, such as ROS-Defender to address discovered vulnerabilities. The three core components of this dissertation were broken into Internal System Defense (Part II), External System Verification (Part III), and Automated Vulnerability Detection (Part IV). These components outlined the key issues with current ROS security tools and provided solutions to address them.

In Part II – Internal System Defense, ROS-Defender and *ROS-FM* were described. These two tools provide defense for ROS against attackers that have gained access to the robot, either through node vulnerabilities or corruption. This area of protection is crucial to maintain the robot's functionality in the inevitable event of software vulnerabilities. In

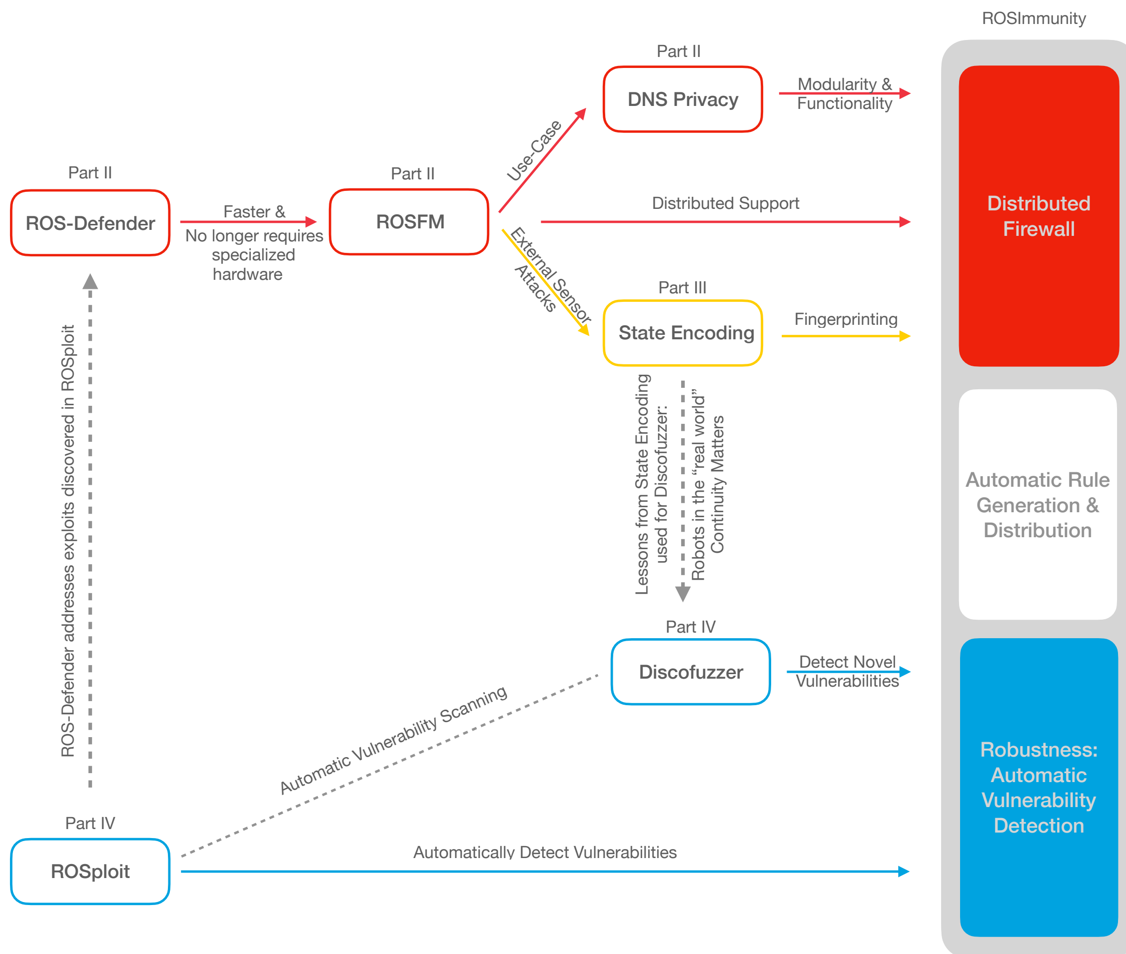


Figure 11.1: The development pathway for *ROS-Immunity*.

both cases, it was clearly shown that these tools provided robustness and reliability to ROS systems.

ROS-Defender and *ROS-FM* provided the framework for the main defensive core of *ROS-Immunity*. The domain-specific language and monitoring components of ROS-Defender were extended into *ROS-FM*, and further extended into *ROS-Immunity* to provide the backbone of the communication system used to maintain consistency between firewalls and grant users control over their security systems. The firewall component of *ROS-Immunity* was patterned off of the original design of *ROS-FM*, with extensions for multiple robots commu-

nicating over a network.

In Part III – External System Verification, the state encoding method was discussed. This method addressed risks in robotic behavior, particularly when the robot was behaving 'correctly' but an attacker distorts its view of the world, and subsequently, its behavior. This chapter addressed sensor attacks, where an attacker feeds malicious data to the robotics sensors, which was an essential absence in the state-of-the-art at that time. The autoencoder framework established in this chapter is a vital component in the design of *ROS-Immunity*. In *ROS-Immunity*, this framework was implemented as a plug-in for *ROS-FM* to extend the firewall capabilities of the final tool.

In Part IV – Automated Vulnerability Detection, *ROSploit* and *DiscoFuzzer* were introduced. Automated vulnerability detection is a key component in *ROS-Immunity*, filling a significant gap in research in the field of ROS security systems at the time. *ROSploit* was established as a tool that ensured to provide users with a vulnerability fingerprint scanner, automatically scanning the users' systems for the presence of actively exploitable, known vulnerabilities. *DiscoFuzzer* provides a method of discovering novel vulnerabilities specific to ROS. Working in conjunction with the automated vulnerability detection systems allow *ROS-Immunity* to continuously update its protections in the face of newly discovered vulnerabilities, including discovering such vulnerabilities itself.

These tools led to the development of *ROS-Immunity*, discussed in completion in Part I – Integrated Security. *ROS-Immunity* is one of the first complete system security systems for ROS, addressing all current security concerns in ROS when working in concert with the native ROS security features, *ROS-Immunity* provides a comprehensive security suite of tools with modular interfaces with support for multiple robot configurations.

ROS-Immunity combines several methods to increase security in ROS systems, discussed here as the development of a modular firewall and an automatic vulnerability detection system. These parts of *ROS-Immunity* were discussed in-depth and connected via their components throughout this dissertation. The novelty of *ROS-Immunity* also lies in developing its communication system, allowing communication between robots to function even if robots are compromised. These three methods provide a complete security system

for ROS, a unique and highly effective solution.

11.0.1 Future Work

As robots continue to evolve, the types of attacks *ROS-Immunity* will have to defend against will evolve alongside them. While *ROS-Immunity*'s modular design and extensible system combine to provide it with some degree of 'future-proofing', it will still require the addition of novel plug-ins to deal with these attacks.

Additionally, with the increasing presence of cloud infrastructure, the types of attacks and scope of trust will have to increase continuously. The field of robotic security may completely transform within the next decade, with it the addition of unique attacks and vulnerabilities that cannot be anticipated.

Development of *ROS-Immunity* will continue to ensure that the tool is up-to-date and maintains the ability to address future attacks and vulnerabilities. The development of a complete security tool is always ongoing; thus, so will the development of this tool.

11.0.2 Closing Remarks

This dissertation clearly outlined the issues, concerns, and solutions for robotic security in the Robotic Operating System. The development of *ROS-Immunity* and its components demonstrated the importance and validity of securing ROS systems and providing a method of doing so. As robots continue to expand more into day-to-day life, *ROS-Immunity* will become an essential component of everyday robot safety.

It is my hope that the field of robotic security does not follow the trends set by the computer security field, where historically, security solutions have been adopted sparsely and slowly, generally only adopted after large failure. Robotic systems are essential in everyday life, and a catastrophic security breach could have consequences far exceeding those of previous breaches. As the robotic security field progresses, it is of utmost importance to increase awareness of the issues and solutions to mitigate them. Vigilance is vital as the stakes of significant security issues in this field are much higher.

References

[1] Henry B Mann and Donald R Whitney. “On a test of whether one of two random variables is stochastically larger than the other”. In: *The annals of mathematical statistics* (1947), pp. 50–60.

[2] Martin A. Fischler and Robert C. Bolles. “Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography”. In: *Commun. ACM* (1981), pp. 381–395.

[3] Steven McCanne and Van Jacobson. “The BSD Packet Filter: A New Architecture for User-level Packet Capture”. In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX’93. San Diego, California: USENIX Association, 1993, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=1267303.1267305.

[4] P. Rocco. “Stability of PID control for industrial robot arms”. In: *IEEE Transactions on Robotics and Automation* 12.4 (Aug. 1996), pp. 606–614. ISSN: 1042-296X. DOI: 10.1109/70.508444.

[5] Robert Weiss. “An approach to Bayesian sensitivity analysis”. In: *Journal of the Royal Statistical Society: Series B (Methodological)* 58.4 (1996), pp. 739–750.

[6] Alma Whitten and J Doug Tygar. “Why Johnny Can’t Encrypt: A Usability Evaluation of PGP 5.0.” In: *USENIX Security Symposium*. Vol. 348. 1999, pp. 169–184.

[7] Edward W Felten and Michael A Schneider. "Timing attacks on web privacy". In: *Proceedings of the 7th ACM conference on Computer and communications security*. 2000, pp. 25–32.

[8] András Vargha and Harold D Delaney. "A critique and improvement of the CL common language effect size statistics of McGraw and Wong". In: *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132.

[9] MA Hernández. "Chebyshev's approximation algorithms and applications". In: *Computers & Mathematics with Applications* 41.3-4 (2001), pp. 433–445.

[10] Jaeyeon Jung et al. "DNS performance and the effectiveness of caching". In: *IEEE/ACM Transactions on networking* 10.5 (2002), pp. 589–603.

[11] Alexander Shapiro. "Monte Carlo sampling methods". In: *Handbooks in operations research and management science* 10 (2003), pp. 353–425.

[12] Konrad Iwanicki, Maarten Van Steen, and Spyros Voulgaris. "Gossip-based clock synchronization for large decentralized systems". In: *IEEE International Workshop on Self-Managed Networks, Systems, and Services*. Springer. 2006, pp. 28–42.

[13] Peter Mell, Karen Scarfone, and Sasha Romanosky. "A complete guide to the common vulnerability scoring system version 2.0". In: *Published by FIRST-forum of incident response and security teams*. Vol. 1. 2007, p. 23.

[14] Fyodor. "Nmap Network Scanning". In: Nmap Project; 12.2.2008 edition (January 1, 2009), 2008. Chap. Nmap Scripting Engine. ISBN: 0-9799587-1-7. URL: `https://nmap.org/book/nse.html`.

[15] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[16] Guy Bruneau and Rick Wanner. "Dns sinkhole". In: *SANS Institute InfoSec Reading Room, Aug* 7 (2010).

[17] Karl Koscher et al. "Experimental security analysis of a modern automobile". In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 447–462.

[18] Stephen Checkoway et al. “Comprehensive experimental analyses of automotive attack surfaces.” In: *USENIX Security Symposium*. Vol. 4. San Francisco. 2011, pp. 447–462.

[19] Shengyong Chen, Youfu Li, and Ngai Ming Kwok. “Active vision in robotic systems: A survey of recent developments”. In: *The International Journal of Robotics Research* 30.11 (2011), pp. 1343–1377.

[20] Peter Dorfinger, Georg Panholzer, and Wolfgang John. “Entropy Estimation for Real-Time Encrypted Traffic Identification (Short Paper)”. In: *Traffic Monitoring and Analysis*. Ed. by Jordi Domingo-Pascual, Yuval Shavitt, and Steve Uhlig. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 164–171. ISBN: 978-3-642-20305-3.

[21] Carlo Pinciroli et al. “ARGoS: a modular, multi-engine simulator for heterogeneous swarm robotics”. In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2011, pp. 5027–5034.

[22] Ayssam Elkady and Tarek Sobh. “Robotics middleware: A comprehensive literature survey and attribute-based bibliography”. In: *Journal of Robotics* 2012 (2012).

[23] Konstantin Serebryany et al. “AddressSanitizer: A fast address sanity checker”. In: *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*. 2012, pp. 309–318.

[24] Manshi Shukla and Amar Nath Shukla. “Growth of robotics industry early in 21st century”. In: *International Journal of Computational Engineering Research* 2.5 (2012), pp. 1554–1558.

[25] Gerald Steinbauer. “A survey about faults of robots used in robocup”. In: *Robot Soccer World Cup*. Springer. 2012, pp. 344–355.

[26] The Open Networking Foundation. *OpenFlow Switch Specification*. June 2012.

[27] Manuele Brambilla et al. “Swarm robotics: a review from the swarm engineering perspective”. In: *Swarm Intelligence* 7.1 (2013), pp. 1–41.

[28] Aaron Martinez and Enrique Fernández. *Learning ROS for robotics programming*. Packt Publishing Ltd, 2013.

[29] Jarrod McClean et al. "A preliminary cyber-physical security assessment of the robot operating system (ros)". In: *Unmanned Systems Technology XV*. Vol. 8741. International Society for Optics and Photonics. 2013, p. 874110.

[30] Justyna Zander. "Model-based testing for execution algorithms in the simulation of cyber-physical systems". In: *2013 IEEE AUTOTESTCON*. IEEE. 2013, pp. 1–7.

[31] Nicola Bezzo et al. "Attack resilient state estimation for autonomous robotic systems". In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2014, pp. 3692–3698.

[32] Andreas Bihlmaier and Heinz Wörn. "Increasing ROS reliability and safety through advanced introspection capabilities". In: *Informatik 2014*. Ed. by E. Plödereder et al. Bonn: Gesellschaft für Informatik e.V., 2014, pp. 1319–1326.

[33] Sean Gallagher. *Photos of an NSA "upgrade" factory show Cisco router getting implant*. May 2014. URL: https://arstechnica.com/tech-policy/2014/05/photos-of-an-nsa-upgrade-factory-show-cisco-router-getting-implant/.

[34] Dominik Herrmann, Max Maaß, and Hannes Federrath. "Evaluating the security of a DNS query obfuscation scheme for private web surfing". In: *IFIP International Information Security Conference*. Springer. 2014, pp. 205–219.

[35] Mouna Jouini, Latifa Ben Arfa Rabai, and Anis Ben Aissa. "Classification of Security Threats in Information Systems". In: *Procedia Computer Science* 32 (2014). The 5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014), the 4th International Conference on Sustainable Energy Information Technology (SEIT-2014), pp. 489–496. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2014.05.452. URL: http://www.sciencedirect.com/science/article/pii/S1877050914006528.

[36] Jae Kwon. "Tendermint: Consensus without mining". In: *Draft v. 0.6, fall* 1.11 (2014).

[37] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (2014), p. 2.

[38] V. Monajjemi, J. Wawerla, and R. Vaughan. “Drums: A Middleware-Aware Distributed Robot Monitoring System”. In: *2014 Canadian Conference on Computer and Robot Vision*. 2014, pp. 211–218.

[39] J. R. C. Nurse et al. “Understanding Insider Threat: A Framework for Characterising Attacks”. In: *2014 IEEE Security and Privacy Workshops*. May 2014, pp. 214–228. DOI: 10.1109/SPW.2014.38.

[40] R. Toris, C. Shue, and S. Chernova. “Message authentication codes for secure remote non-native client connections to ROS enabled robots”. In: Apr. 2014, pp. 1–6. DOI: 10.1109/TePRA.2014.6869141.

[41] Tamara Bonaci et al. “To make a robot secure: An experimental analysis of cyber security threats against teleoperated surgical robots”. In: *arXiv preprint arXiv:1504.04339* (2015).

[42] S. Bortzmeyer. *DNS Privacy Considerations*. RFC 7626. RFC Editor, Aug. 2015. URL: https://tools.ietf.org/html/rfc7626.

[43] François Chollet. *keras*. https://github.com/fchollet/keras. 2015.

[44] S. Kato et al. “An Open Approach to Autonomous Vehicles”. In: *IEEE Micro* 35.6 (Nov. 2015), pp. 60–68. ISSN: 0272-1732. DOI: 10.1109/MM.2015.133.

[45] Dae Wook Kim and Junjie Zhang. “You are how you query: Deriving behavioral fingerprints from DNS traffic”. In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2015, pp. 348–366.

[46] Yong Li and Min Chen. “Software-defined network function virtualization: A survey”. In: *IEEE Access* 3 (2015), pp. 2542–2553.

[47] Santiago Morante, Juan G Victores, and Carlos Balaguer. “Cryptobotics: Why robots need cyber safety”. In: *Frontiers in Robotics and AI* 2 (2015), p. 23.

[48] Jonathan Petit et al. *Remote Attacks on Automated Vehicles Sensors: Experiments on Camera and LiDAR*. Tech. rep. OnBoard Security Inc., 2015.

[49] Francesca Saglietti et al. "Model-based design and testing of decisional autonomy and cooperation in cyber-physical systems". In: *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*. IEEE. 2015, pp. 479–483.

[50] Artur Saudabayev and Huseyin Atakan Varol. "Sensors for robotic hands: A survey of state of the art". In: *IEEE Access* 3 (2015), pp. 1765–1782.

[51] Michal Zalewski. "american fuzzy lop technical "whitepaper","". In: *URL: http://lcamtuf. coredump. cx/afl/technical_details. txt* (2015).

[52] Homa Alemzadeh et al. "Targeted attacks on teleoperated surgical robots: Dynamic model-based detection and mitigation". In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2016, pp. 395–406.

[53] Andreas Bihlmaier, Matthias Hadlich, and Heinz Wörn. "Advanced ROS Network Introspection (ARNI)". In: *Robot Operating System (ROS): The Complete Reference (Volume 1)*. Ed. by Anis Koubaa. Cham: Springer International Publishing, 2016, pp. 651–670.

[54] Yuting Chen et al. "Coverage-directed differential testing of JVM implementations". In: *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2016, pp. 85–99.

[55] Drew Davidson et al. "Controlling UAVs with Sensor Input Spoofing Attacks". In: *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. Austin, TX: USENIX Association, 2016. URL: https://www.usenix.org/conference/woot16/workshop-program/presentation/davidson.

[56] Bernhard Dieber et al. "Application-level security for ROS-based applications". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2016, pp. 4477–4482.

[57] Fadri Furrer et al. "Robot Operating System (ROS): The Complete Reference (Volume 1)". In: ed. by Anis Koubaa. Cham: Springer International Publishing, 2016. Chap. RotorS—A Modular Gazebo MAV Simulator Framework, pp. 595–625. ISBN: 978-3-319-26054-9. DOI: 10.1007/978-3-319-26054-9_23. URL: http://dx.doi.org/10.1007/978-3-319-26054-9_23.

[58] Z. Hu et al. *Specification for DNS over Transport Layer Security (TLS)*. [Online; accessed 01-October-2019]. 2016. URL: https://www.rfc-editor.org/rfc/rfc7858.txt.

[59] Jefferson Ryan Medel and Andreas E. Savakis. "Anomaly Detection in Video Using Predictive Convolutional Long Short-Term Memory Networks". In: *CoRR* abs/1612.00390 (2016).

[60] S Pavithirakini et al. "Improve the Capabilities of Wireshark as a tool for Intrusion Detection in DOS Attacks". In: *International Journal of Scientific and Research Publications* 6.4 (2016), pp. 378–384.

[61] Christophe Ponsard, Philippe Massonet, and Gautier Dallons. "Co-engineering security and safety requirements for cyber-physical systems". In: *ERCIM NEWS* 106 (2016), pp. 45–46.

[62] André Santos et al. "A framework for quality assessment of ROS repositories". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2016, pp. 4491–4496.

[63] Nick Stephens et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In: *NDSS*. Vol. 16. 2016. 2016, pp. 1–16.

[64] Ruffin White, Henrik I. Christensen, and Morgan Quigley. "SROS: Securing ROS over the wire, in the graph, and through the kernel". In: *CoRR* abs/1611.07060 (2016). arXiv: 1611.07060. URL: http://arxiv.org/abs/1611.07060.

[65] Chen Yan, Wenyuan Xu, and Jianhao Liu. "Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle". In: *DEF CON* 24 (2016).

[66] Gilberto Bertin. “XDP in practice: integrating XDP into our DDoS mitigation pipeline”. In: *Technical Conference on Linux Networking, Netdev*. Vol. 2. 2017.

[67] Jahshan Bhatti and Todd E. Humphreys. “Hostile Control of Ships via False GPS Signals: Demonstration and Detection”. In: *Navigation* 64 (Mar. 2017), pp. 51–66. DOI: 10.1002/navi.183.

[68] B. Breiling, B. Dieber, and P. Schartner. “Secure communication for the robot operating system”. In: Apr. 2017, pp. 1–6. DOI: 10.1109/SYSCON.2017.7934755.

[69] Yong Shean Chong and Yong Haur Tay. “Abnormal Event Detection in Videos Using Spatiotemporal Autoencoder”. In: *Advances in Neural Networks - ISNN 2017 - 14th International Symposium, ISNN 2017, Sapporo, Hakodate, and Muroran, Hokkaido, Japan, June 21-26, 2017, Proceedings, Part II*. 2017, pp. 189–196.

[70] G. W. Clark, M. V. Doran, and T. R. Andel. “Cybersecurity issues in robotics”. In: *2017 IEEE Conference on Cognitive and Computational Aspects of Situation Management (CogSIMA)*. Mar. 2017, pp. 1–5. DOI: 10.1109/COGSIMA.2017.7929597.

[71] Christopher Crick et al. “Rosbridge: Ros for non-ros users”. In: *Robotics Research*. Springer, 2017, pp. 493–504.

[72] Bernhard Dieber et al. “Security for the Robot Operating System”. In: *Robotics and Autonomous Systems* 98 (2017), pp. 192–203. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2017.09.017. URL: http://www.sciencedirect.com/science/article/pii/S0921889017302762.

[73] Mahmoud Hashem Eiza and Qiang Ni. “Driving with sharks: Rethinking connected vehicles with vehicle cybersecurity”. In: *IEEE Vehicular Technology Magazine* 12.2 (2017), pp. 45–51.

[74] Matt Fleming. “A thorough introduction to eBPF”. In: *Linux Weekly News* (2017).

[75] Pinyao Guo et al. “Exploiting Physical Dynamics to Detect Actuator and Sensor Attacks in Mobile Robots”. In: *arXiv* (Aug. 2017). URL: http://arxiv.org/abs/1708.01834.

[76] Qiyi He, Xiaolin Meng, and Rong Qu. “Survey on cyber security of CAV”. In: *2017 Forum on Cooperative Positioning and Service (CPGPS)*. IEEE. 2017, pp. 351–354.

[77] Neminath Hubballi and Nikhil Tripathi. “An event based technique for detecting spoofed IP packets”. In: *Journal of Information Security and Applications* 35 (2017), pp. 32–43. ISSN: 2214-2126. DOI: https://doi.org/10.1016/j.jisa.2017.04.001. URL: http://www.sciencedirect.com/science/article/pii/S2214212617301692.

[78] Celine Irvene et al. “HoneyBot: A honeypot for robotic systems”. In: *Proceedings of the IEEE* 106.1 (2017), pp. 61–70.

[79] Se-Yeon Jeong et al. “A Study on ROS Vulnerabilities and Countermeasure”. In: *Proceedings of the Companion of the 2017 ACM/IEEE International Conference on Human-Robot Interaction*. HRI ’17. Vienna, Austria: ACM, 2017, pp. 147–148. ISBN: 978-1-4503-4885-0. DOI: 10.1145/3029798.3038437. URL: http://doi.acm.org/10.1145/3029798.3038437.

[80] D. Wing K. T. Reddy and P. Patil. *DNS over Datagram Transport Layer Security (DTLS)*. Tech. rep. [Online; accessed 01-October-2019]. 2017. URL: https://www.rfc-editor.org/rfc/rfc8094.txt.

[81] CG Leela Krishna and Robin R Murphy. “A review on cybersecurity vulnerabilities for unmanned aerial vehicles”. In: *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE. 2017, pp. 194–199.

[82] Francisco J RodrıFIX ME!!!!guez Lera et al. “Cybersecurity of robotics and autonomous systems: Privacy and safety”. In: *Robotics-legal, ethical and socioeconomic impacts* (2017).

[83] Theofilos Petsios et al. “Nezha: Efficient domain-independent differential testing”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 615–632.

[84] D. Quarta et al. “An Experimental Security Analysis of an Industrial Robot Controller”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 268–286.

[85] R. Rahimi et al. “An Industrial Robotics Application with Cloud Computing and High-Speed Networking”. In: *IEEE International Conference on Robotic Computing (IRC)*. 2017, pp. 44–51.

[86] Sanjay Rawat et al. “VUzzer: Application-aware Evolutionary Fuzzing.” In: *NDSS*. Vol. 17. 2017, pp. 1–14.

[87] André Santos et al. “Mining the usage patterns of ROS primitives”. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2017, pp. 3855–3860.

[88] Hocheol Shin et al. “Illusion and Dazzle: Adversarial Optical Channel Exploits Against Lidars for Automotive Applications”. In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. 2017, pp. 445–467.

[89] Ryu Project Team. *RYU SDN Framework*. 2017. URL: https://osrg.github.io/ryu/.

[90] H. Umari and S. Mukhopadhyay. “Autonomous robotic exploration based on multiple rapidly-exploring randomized trees”. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2017, pp. 1396–1402. DOI: 10.1109/IROS.2017.8202319.

[91] Johannes Wienke and Sebastian Wrede. “Results of the survey: failures in robotics and intelligent systems”. In: *arXiv preprint arXiv:1708.07379* (2017).

[92] Khalil M Ahmad Yousef et al. “Analyzing cyber-physical threats on robotic platforms”. In: *Sensors* 18.5 (2018), p. 1643.

[93] Khalil M. Ahmad Yousef et al. “Analyzing cyber-physical threats on robotic platforms”. In: *Sensors (Switzerland)* 18.5 (May 2018). ISSN: 14248220. DOI: 10.3390/s18051643.

[94] Advanced Threat Analytics. *New Research from Advanced Threat Analytics Finds MSSP Incident Responders Overwhelmed by False-Positive Security Alerts*. June 2018. URL: %7Bhttps://prn.to/2uTiaK6%7D.

[95] Mordechai Ben-Ari and Francesco Mondada. “Robots and their applications”. In: *Elements of Robotics*. Springer, 2018, pp. 1–20.

[96] Agnes Cameron, Mark Payne, and Bruno Prela. “Research and Implementation of Multiple Blockchain Byzantine Secure Consensus Protocols for Robot Swarms”. In: (2018).

[97] Gabriel B. Cavallari, Leonardo Sampaio Ferraz Ribeiro, and Moacir Antonelli Ponti. “Unsupervised representation learning using convolutional and stacked auto-encoders: a domain and cross-domain feature space analysis”. In: *CoRR* abs/1811.00473 (2018).

[98] Peng Chen and Hao Chen. “Angora: Efficient fuzzing by principled search”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 711–725.

[99] Hongjun Choi et al. “Detecting Attacks Against Robotic Vehicles”. In: New York, New York, USA: Association for Computing Machinery (ACM), Oct. 2018, pp. 801–816. DOI: 10.1145/3243734.3243752.

[100] Vincenzo DiLuoffo, William R Michalson, and Berk Sunar. “Robot Operating System 2: The need for a holistic security approach to robotic architectures”. In: *International Journal of Advanced Robotic Systems* 15.3 (2018), p. 1729881418770011.

[101] Eduardo Castelló Ferrer. “The blockchain: a new framework for robotic swarm systems”. In: *Proceedings of the future technologies conference*. Springer. 2018, pp. 1037–1058.

[102] Alberto Giaretta, Michele De Donno, and Nicola Dragoni. “Adding Salt to Pepper: A Structured Security Assessment over a Humanoid Robot”. In: *CoRR* abs/1805.04101 (2018). arXiv: 1805.04101. URL: http://arxiv.org/abs/1805.04101.

[103] P. Guo et al. "RoboADS: Anomaly Detection Against Sensor and Actuator Misbehaviors in Mobile Robots". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2018, pp. 574–585.

[104] Ritwik Gupta et al. "Open Problems in Robotic Anomaly Detection". In: *arXiv preprint arXiv:1809.03565* (2018).

[105] P. E. Hoffman and P. McManus. *DNS Queries over HTTPS (DoH)*. [Online; accessed 01-October-2019]. 2018. URL: https://www.rfc-editor.org/rfc/rfc8484.txt.

[106] Toke Høiland-Jørgensen et al. "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel". In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '18. Heraklion, Greece, 2018, pp. 54–66. ISBN: 978-1-4503-6080-7.

[107] Saurabh Jain and Rajesh Doriya. "Security Issues and Solutions in Cloud Robotics: A Survey". In: *International Conference on Next Generation Computing Technologies*. Springer. 2018, pp. 64–76.

[108] Vivek Jain et al. "TIFF: using input type inference to improve fuzzing". In: *Proceedings of the 34th Annual Computer Security Applications Conference*. 2018, pp. 505–517.

[109] P. Kapoor, A. Vora, and K. Kang. "Detecting and Mitigating Spoofing Attack Against an Automotive Radar". In: *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)*. 2018, pp. 1–6.

[110] Shinpei Kato et al. "Autoware on board: Enabling autonomous vehicles with embedded systems". In: *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE. 2018, pp. 287–296.

[111] Azfar Khalid et al. "Security framework for industrial collaborative robotic cyber-physical systems". In: *Computers in Industry* 97 (2018), pp. 132–145.

[112] Jongkil Kim et al. "Security and performance considerations in ros 2: A balancing act". In: *arXiv preprint arXiv:1809.09566* (2018).

[113] George Klees et al. "Evaluating fuzz testing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 2123–2138.

[114] Daniel M Lofaro et al. "Extending the life of legacy robots: MDS-Ach, a real-time, process based, networked, secure middleware based on the x-Ach methodology". In: *2018 IEEE 15th International Workshop on Advanced Motion Control (AMC)*. IEEE. 2018, pp. 72–77.

[115] Yanxiang Mao. "Detailed Power Measurement with Arm Embedded Boards". In: (2018).

[116] Francisco Martín, Enrique Soriano, and José M. Cañas. "Quantitative analysis of security in distributed robotic frameworks". In: *Robotics and Autonomous Systems* 100 (2018), pp. 95–107. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2017.11.002. URL: http://www.sciencedirect.com/science/article/pii/S0921889017303044.

[117] Michael James Michaud, Thomas Dean, and Sylvain P Leblanc. "Attacking OMG data distribution service (DDS) based real-time mission critical distributed systems". In: *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE. 2018, pp. 68–77.

[118] Justin Miller, Andrew B Williams, and Debbie Perouli. "A Case Study on the Cybersecurity of Social Robots". In: *Companion of the 2018 ACM/IEEE International Conference on Human-Robot Interaction*. 2018, pp. 195–196.

[119] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. "T-Fuzz: fuzzing by program transformation". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 697–710.

[120] Ishaani Priyadarshini. "Cyber security risks in robotics". In: *Cyber Security and Threats: Concepts, Methodologies, Tools, and Applications*. IGI Global, 2018, pp. 1235–1250.

[121] Mike Richardson and Michael Tyrrell. *Time to join the digital dots*. July 2018. URL: https://www.aero-mag.com/meggitt-applied-research-technology-group-data-capture/.

[122] Francisco J. Rodríguez-Lera et al. "Message Encryption in Robot Operating System: Collateral Effects of Hardening Mobile Robots". In: *Frontiers in ICT* 5 (2018), p. 2.

[123] Francisco J. Rodríguez-Lera et al. "Message Encryption in Robot Operating System: Collateral Effects of Hardening Mobile Robots". In: *Frontiers in ICT* 5 (2018), p. 2. ISSN: 2297-198X. DOI: 10.3389/fict.2018.00002. URL: https://www.frontiersin.org/article/10.3389/fict.2018.00002.

[124] Felipe Roman, Alexandre M Amory, and Renan Maidana. "Overseer: A Multi Robot Monitoring Infrastructure." In: *ICINCO (1)*. 2018, pp. 151–158.

[125] André Santos, Alcino Cunha, and Nuno Macedo. "Property-based testing for the robot operating system". In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 2018, pp. 56–62.

[126] OMG Std. *DDS Security Version 1.1*. 2018.

[127] Volker Strobel, Eduardo Castelló Ferrer, and Marco Dorigo. "Managing byzantine robots via blockchain technology in a swarm robotics collective decision making scenario". In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 2018, pp. 541–549.

[128] V Mayoral Vilches et al. "Introducing the robot security framework (rsf), a standardized methodology to perform security assessments in robotics". In: *arXiv preprint arXiv:1806.04042* (2018).

[129] Mathieu Xhonneux, Fabien Duchene, and Olivier Bonaventure. "Leveraging eBPF for Programmable Network Functions with IPv6 Segment Routing". In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '18. Heraklion, Greece, 2018, pp. 67–72. ISBN: 978-1-4503-6080-7.

[130] Insu Yun et al. “{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing”. In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 745–761.

[131] alexa. *Top 500 sites on the web*. 2019. URL: https://www.alexa.com/topsites.

[132] Farshad Arvin et al. “Mona: an affordable open-source mobile robot for education and research”. In: *Journal of Intelligent & Robotic Systems* 94.3-4 (2019), pp. 761–775.

[133] Cornelius Aschermann et al. “REDQUEEN: Fuzzing with Input-to-State Correspondence.” In: *NDSS*. Vol. 19. 2019, pp. 1–15.

[134] AWS-Robotics. *Health Metric Collector*. https://github.com/aws-robotics/health-metrics-collector-ros1. 2019.

[135] AWS-Robotics. *ROS2-SecTest*. https://github.com/aws-robotics/ROS2-SecTest. 2019.

[136] *BPF and XDP Reference Guide*. https://docs.cilium.io/en/v1.4/bpf/. [Online; accessed 30-August-2019]. 2019.

[137] Gianluca Caiazza, Ruffin White, and Agostino Cortesi. “Enhancing security in ROS”. In: *Advanced Computing and Systems for Security*. Springer, 2019, pp. 3–15.

[138] Edward Cone and James Lambert. *How Robots Change the World - What automation really means for jobs, productivity and regions*. June 2019. URL: https://www.oxfordeconomics.com/recent-releases/how-robots-change-the-world.

[139] dnscrypt. *dnscrypt*. [Online; accessed 01-October-2019]. 2019. URL: https://dnscrypt.info/.

[140] EFF. *Panopticlick 3.0*. 2019. URL: https://panopticlick.eff.org/.

[141] Pablo Estefo et al. “The Robot Operating System: Package reuse and community dynamics”. In: *Journal of Systems and Software* 151 (2019), pp. 226–242.

[142] Eduardo Castelló Ferrer et al. “Secure and secret cooperation of robotic swarms by using Merkle trees”. In: *arXiv preprint arXiv:1904.09266* (2019).

[143] A Gasparetto and L Scalera. “A brief history of industrial robotics in the 20th century”. In: *Advances in Historical Studies* 8.1 (2019), pp. 24–35.

[144] Brendan Gregg. *eBPF*. [Online; accessed 15-October-2019]. 2019. URL: http://www.brendangregg.com/ebpf.html.

[145] IOVISOR. *BPF Compiler Collection (BCC)*. https://github.com/iovisor/bcc. 2019.

[146] Taegyu Kim et al. “RVFUZZER: finding input validation bugs in robotic vehicles through control-guided testing”. In: *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 2019, pp. 425–442.

[147] Valentin Jean Marie Manès et al. “The art, science, and engineering of fuzzing: A survey”. In: *IEEE Transactions on Software Engineering* (2019).

[148] Netflix. *Vector*. https://github.com/Netflix/vector. 2019.

[149] Deborah Petrara. *The Rise of ROS: Nearly 55% of total commercial robots shipped in 2024 Will Have at Least One Robot Operating System package*. 2019. URL: https://www.bloomberg.com/press-releases/2019-05-16/the-rise-of-ros-nearly-55-of-total-commercial-robots-shipped-in-2024-will%20%5C%5C%20-have%20-at%20-%20least%20-one%20-%20robot-%20operating-%20system-%20package.

[150] Marc Pichler, Bernhard Dieber, and Martin Pinzger. “Can i depend on you? Mapping the dependency and quality landscape or ROS packages”. In: *Proceedings of the 3rd International Conference on Robotic Computing*. IEEE, Feb. 2019.

[151] Marcello Pogliani et al. “Security of controlled manufacturing systems in the connected factory: the case of industrial robots”. In: *Journal of Computer Virology and Hacking Techniques* 15.3 (2019), pp. 161–175.

[152] “Real-time attack detection on robot cameras: A self-driving car application”. In: *2019 Third IEEE International Conference on Robotic Computing (IRC)*. IEEE. 2019, pp. 102–109.

[153] Sean Rivera, Sofiane Lagraa, and Radu State. “ROSploit: Cybersecurity Tool for ROS”. In: *3rd IEEE International Conference on Robotic Computing, IRC 2019, Naples, Italy, February 25-27, 2019*. 2019, pp. 415–416.

[154] Sean Rivera et al. “Auto-encoding Robot State against Sensor Spoofing Attacks”. In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE. 2019, pp. 252–257.

[155] Sean Rivera et al. “Ros-defender: Sdn-based security policy enforcement for robotic applications”. In: *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2019, pp. 114–119.

[156] André Santos, Alcino Cunha, and Nuno Macedo. “Static-time extraction and analysis of the ROS computation graph”. In: *2019 Third IEEE International Conference on Robotic Computing (IRC)*. IEEE. 2019, pp. 62–69.

[157] Sandra Siby et al. “Encrypted DNS -¿ Privacy? A Traffic Analysis Perspective”. In: *CoRR* abs/1906.09682 (2019). arXiv: 1906.09682. URL: http://arxiv.org/abs/1906.09682.

[158] Junjie Wang et al. “Superion: Grammar-aware greybox fuzzing”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 724–735.

[159] Wei You et al. “Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 769–786.

[160] Lei Zhao et al. “Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing.” In: *NDSS*. 2019.

[161] Mary B Alatise and Gerhard P Hancke. “A Review on Challenges of Autonomous Mobile Robot and Sensor Fusion Methods”. In: *IEEE Access* 8 (2020), pp. 39830–39846.

[162] Faouzi Amrouche et al. “Intrusion detection on robot cameras using spatio-temporal autoencoders: A self-driving car application”. In: *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*. IEEE. 2020, pp. 1–5.

[163] Abdullahi Chowdhury et al. “Attacks on Self-Driving Cars and Their Countermeasures: A Survey”. In: *IEEE Access* 8 (2020), pp. 207308–207342.

[164] Bernhard Dieber et al. “Penetration Testing ROS”. In: *Robot Operating System (ROS)*. Springer, 2020, pp. 183–225.

[165] G Lacava et al. “Current Research Issues on Cyber security in Robotics”. In: (2020).

[166] LLVM-Developers. *MemorySanitizer*. 2020. URL: `http://clang.llvm.org/docs/MemorySanitizer.html` (visited on 02/17/2020).

[167] LLVM-Developers. *UndefinedBehaviorSanitizer*. 2020. URL: `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html` (visited on 02/17/2020).

[168] VıFIX ME!!!!ctor Mayoral-Vilches et al. “Can ROS be used securely in industry? Red teaming ROS-Industrial”. In: *arXiv preprint arXiv:2009.08211* (2020).

[169] Sean Rivera, Antonio Ken Iannillo, and Radu State. “DiscoFuzzer: Discontinuity-based Vulnerability Detector for Robotic Systems”. 2020.

[170] Sean Rivera, Antonio Ken Iannillo, and Radu State. “ROS-Immunity: Integrated Approach for the Security of ROS-enabled Robotic Systems”. 2020.

[171] Sean Rivera et al. “Leveraging eBPF to preserve user privacy for DNS, DoT, and DoH queries”. In: *Proceedings of the 15th International Conference on Availability, Reliability and Security*. 2020, pp. 1–10.

[172] ROS-Industrial. *ROS-Industrial*. 2020. URL: `https://rosindustrial.org/` (visited on 02/17/2020).

[173] ROS.org. *Automatic Testing with ROS*. 2020. URL: `http://wiki.ros.org/Quality/Tutorials/UnitTesting` (visited on 02/17/2020).

[174] ROS.org. *rostopic*. `http://wiki.ros.org/rostopic`. 2020.

[175] Chris Swierczewski and Olivier Verdier. *Pychebfun*. https://github.com/pychebfun/pychebfun. 2020.

[176] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* (2020). DOI: https://doi.org/10.1038/s41592-019-0686-2.

[177] Sean Rivera et al. “Fast Monitoring for the Robotic Operating System”. In: *Proceedings of the 25th International Conference on Engineering of Complex Computer Systems*. 2021.

[178] *”ZODIAC WX”*. https://northboundnetworks.com/products/zodiac-wx. Accessed: 2018-06-02.

[179] *AMIGO*. https://robots.ros.org/amigo/. Accessed: 2019-05-19.

[180] *Apparmor*. https://wiki.ubuntu.com/AppArmor. Accessed: 2018-03-27.

[181] Eric Chan-Tin et al. “The frog-boiling attack: Limitations of anomaly detection for secure network coordinate systems”. In: Springer.

[182] Jonathan Chu. *Army Robotics in the Military*. https://insights.sei.cmu.edu/sei_blog/2017/06/army-robotics-in-the-military.html. Accessed: 2019-05-19.

[183] Borkmann Daniel. “On getting tc classifier fully programmable with cls bpf”. In: *Proceedings of netdev*. 2016.

[184] Fyodor. *The Art of Port Scanning*. https://nmap.org/p51-11.html. Accessed: 2018-06-29.

[185] *Go versus Python 3 fastest programs*. https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/go-python3.html. [Online; accessed 30-August-2019].

[186] *Hacking a MiR Robot with ROSPenTo*. URL: https://bernharddieber.com/post/mir-hacking-video/.

[187] Jia Cheng Han and Zhi Quan Zhou. “Metamorphic Fuzz Testing of Autonomous Vehicles”. In: ().

[188] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* (), pp. 1735–1780.

[189] *Husky UNMANNED GROUND VEHICLE*. https://www.clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/. Accessed: 2018-09-27.

[190] N. Koenig and A. Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In:

[191] *Metasploit: The world's most used penetration testing framework*. https://www.metasploit.com/. Accessed: 2018-06-29.

[192] *Multi-Robot Systems with ROS Lesson 1*. u.cs.biu.ac.il/~yehoshr1/89-689/ROS_Multi_Lesson1.pptx. Accessed: 2018-06-29.

[193] Open Source Robotics Foundation, Inc. *ROS 2 Design*. http://design.ros2.org/. [Online; accessed 1-June-2020].

[194] ROS Project. *TCPROS*. http://wiki.ros.org/ROS/TCPROS. Accessed: 2019-08-30.

[195] ROS Project. *UDPROS*. http://wiki.ros.org/ROS/UDPROS. Accessed: 2019-08-30.

[196] Robotics Business Review. *Global Spending on Robots, Drones to Top $115B in 2019, Says IDC)*. Tech. rep. [Online; accessed 30-August-2019].

[197] *ROS Parameter Server*. http://wiki.ros.org/Parameter\%20Server. Accessed: 2018-03-27.

[198] *ROSBAG*. http://wiki.ros.org/rosbag. Accessed: 2018-03-27.

[199] Mayu Sakurada and Takehisa Yairi. "Anomaly Detection Using Autoencoders with Nonlinear Dimensionality Reduction". In: DOI: 10.1145/2689746.2689747. URL: http://doi.acm.org/10.1145/2689746.2689747.

[200] *SROS*. http://wiki.ros.org/SROS. Accessed: 2018-03-27.

[201] TensorFlow. *TensorFlow*. https://www.tensorflow.org/.

[202] *Turtlebot3 Overview*. http://emanual.robotis.com/docs/en/platform/turtlebot3/overview/. Accessed: 2018-06-29.

[203] *Turtlebot3 Waffle*. https://www.generationrobots.com/fr/402716-turtlebot-3-waffle.html. Accessed: 2018-08-02.

[204] *Using Simulated Robot in Robot Studio*. http://wiki.ros.org/abb/Tutorials/RobotStudio. Accessed: 2018-09-27.

[205] Pascal Vincent et al. "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion". In: ().

[206] Dave Watson. "KTLS: Linux Kernel Transport Layer Security". In: *netdevconf*. URL: https://netdevconf.info/1.2/papers/ktls.pdf.

[207] *We're Building an Open Source Self-Driving Car*. https://github.com/udacity/self-driving-car. Accessed: 2018-09-27.

[208] *Robots and robotic devices — Vocabulary*. Standard. Geneva, CH: International Organization for Standardization, Mar. 2012.

Appendices