ROS-Defender: SDN-based Security Policy Enforcement for Robotic Applications

Sean Rivera, Sofiane Lagraa, Radu State
SnT, University of Luxembourg, Luxembourg
firstname.lastname@uni.lu

Cristina Nita-Rotaru,
Northeastern University
c.nitarotaru@northeastern.edu

Sheila Becker
Defence Directorate, Luxembourg
sheila.becker@mae.etat.lu

***Abstract*—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.**

***Keywords*—Robotics, ROS, Security, SDN, OpenVswitch**

I. Introduction

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 [23], consumer and commercial [17], self-driving cars [21], and military [13].

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*, relying ultimately on an unsecure network. Unfortunately, ROS does not provide any security features [22], [19], [24]. Recent work highlighted a number of security threats against ROS. One attack showed how an attacker was able to move a robot to another location [22] through an unauthorized publish where the attacker forged false localization messages and injected them in the network. Another attack showed how an attacker could immobilize a robot [22] by stopping the component and sensor that controls the robot movement. Finally, it was shown how an attacker could compromise privacy by exploiting faulty APIs to obtain access to camera and microphones [17].

Recent efforts have been made in order to add security features to ROS such as TLS and DTLS [12] for secure communication between nodes, web tokens for achieving secure authentication for remote access [26], and cryptographic methods that ensure data confidentiality and integrity [14]. The ROS group has also begun work on SROS [28], a security suite for ROS systems, still highly experimental and not fully implemented into the core ROS framework. Current solutions for security of ROS do not provide protection against 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.

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 dynamically enforce such policies.

In this paper, we focus on securing ROS by leveraging SDN to control per-flow policies, and we augment per-flow control, with a per-topic control, to accommodate ROS semantics. We create two security applications on top of this SDN framework: a firewall and a monitoring-watch. Both applications communicate with the SDN network to react to observed traffic and detect either violations of existing policies, or new attacks and adjust the policies. Finally, we define ROS-Policy-Language, a policy language for ROS applications, that allows 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. The application we demonstrate enable an intrusion-prevention approach and are also able to detect and react to compromised applications.

II. Security of ROS

In this section, we provide an overview of ROS and discuss current security services offered and their limitations.

A. ROS Overview

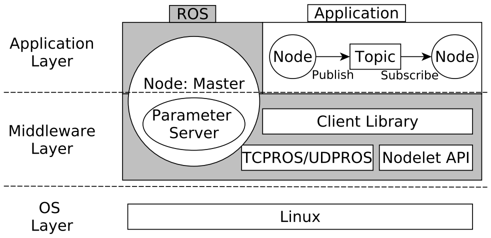

Fig. 1: ROS architecture

ROS [3] is a meta-operating system framework for developing robotic systems (see Figure 1). It 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. They facilitate the communication between nodes based on two abstractions: topics and services. All processes run on top of a UNIX operating system.

ROS master and parameter server. The master node is the main communication hub that tracks all the offered topics and services and maintains a dictionary mapping of the location of all nodes, which nodes are providing which topics/services, and what ports those topics and nodes are located on. The parameter server is a shared repository between nodes. Both the master and the parameter server are implemented using XMLRPC, a stateless HTTP-based protocol.

ROS nodes. Each node is designed to be a self-contained process to control part of the robot's operation and must implement several shared components in order to integrate with the rest of the system. These components include the ROS slave API and the internal ROS protocol and ROS command line interface. The ROS slave API handles all interactions with the ROS master as well as with other ROS nodes.

Topics. Each topic is defined as a uni-directional, 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. At that point, 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, 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.

Services. 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 with and inform the master of the service's location. As with topics, services are located on ephemeral ports by default but can be located at a specific port if desired.

Middleware layer. 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. 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*.

B. Security of ROS

Recent security efforts: SROS. A recent effort into addressing security concerns for ROS is SROS, an experimental security suite designed to harden ROS systems against several common classes of attacks [6]. It is structured around three levels on security concepts; Transport Security, Access Control, and Process Profiles [28]. 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 are unable to 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[6]. 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. [1] 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 as well as what actions the process is allowed to preform. AppArmor confinement is provided via profiles loaded into the kernel, typically on boot.

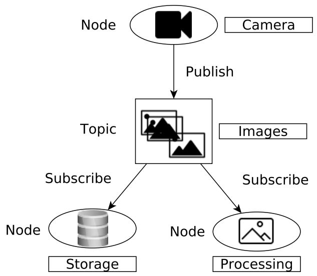

Fig. 2: Example of ROS

Limitations of SROS. 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 reliably maintain the static rules. 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 normal 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.

III. Adversarial Model

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

Independent Assumptions We assume that the master node acts like a certificate authority (CA) or a domain controller and is the root of trust for the system. We also assume that the master node is not compromised before the system is started. We assume 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 it normally has access to. We assume that the ROS middleware is secure from exploits. We understand this might not the case, and consider this grounds for future work. We assume that the underlying Linux system is secure and that best practices are taken in the design of the system such that a compromised ROS node cannot compromise the Linux system.

Attacker Model for ROS Deployment There are three types of communication: node-to-node, node-master, or node-parameter server communication. Remember that in ROS all communication is unprotected, 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. We assume that the attacker shares a network, and thus 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.

Attacker Model for SROS Deployment We assume 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. We assume 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 exhaustive rules for each node to be deployed. Additionally, the whole system has to be redeployed when a rule is updated. As such, we assume that most developers will only implement access control around critical nodes, for ease of use.

IV. ROS-Defender Design

In this section, we describe ROS-Defender. Our system consists of three components: (1) ROSWatch, an anomaly detection system, (2) ROSDN, a prevention system, and (3) ROS-Policy-Language, a policy language. We first describe our design goals and give an overview of ROS-Defender, then describe in detail the main components.

A. 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 a 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 the use of access control and application profiling – techniques proposed by SROS – prevents 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 security of ROS do not provide protection against 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.

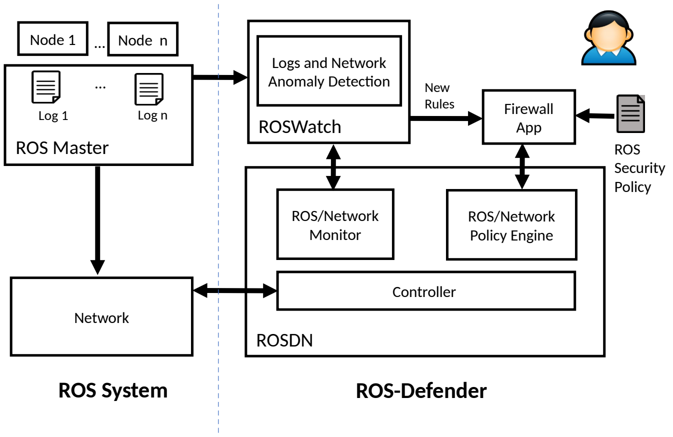

Fig. 3: ROS-Defender Design

Our solution is built with several design goals in mind.

- Provide network level granularity protection. We target the network communication as the granularity of policies we would like to enforce and being able to enforce policies on a per port basis.
- Allow for instant updates to the policy rules and creation of dynamic rules. We would like to be able to support policy changes without system reboot; also be able to dynamically learn new rules that can prevent future attacks.
- 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.
- Support both ROS and SROS. We would like a solution that does not require API changes and provides protection for both ROS, and ROS running with the security extensions provided by SROS.

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 dynamically control network flows at Layer 2, in a network. However, as the policies that we want to enforce are application-layer policies, a mapping between these and the corresponding network layer graph will be needed.

Our approach. Figure 3 shows the design of our system, ROS-Defender. We take an intrusion prevention approach where (1) a user-defined policy is enforced at network level with the help of ROSDN, which replaces the normal ROS network communication with a 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 automatically update the policy and prevent future attacks. 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 part of the developer and it provides security benefits to both ROS and SROS. Note that both ROSDN and ROSWatch can be used independently, but together they provide a much stronger 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.

In Figure 3, we show a firewall application that can be run on our system. The user defines a set of firewall rules using familiar ROS terminology. The ROSDN controller is able to understand this terminology through the use of our proposed domain specific language. ROSDN treats the firewall as a constantly 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 we describe the main components of ROS-Defender.

B. ROSDN

One component of our approach is to replace the network switch with our SDN version in order to access all communication. The ROS middleware is centered around using existing IP communications, as such adding security though the use of SDN is a natural fit. By using SDN we can have a fine-grained control over the network communications that function as the backbone for ROS. We refer to the SDN component 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 our ROS-Policy-Language (see Figure 3). The ROS/Network Monitor maintains an internal state representation of the ROS system, tracking which open ports belong to which nodes, as well as which open ports belong to which topics and services. The ROS Policy Engine translates our domain specific rules language into SDN understandable context, referred to as network-level policies. It leverages *textX* in order to define the language and the watchdog library in order to check the file for changes.

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 correctly filter new connections. 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.

C. 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. Our 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. We construct rules in the following ways:

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

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

D. ROS-Policy-Language

In order to grant ROS developers easy access to the underlying components of the SDN system, we have devised a policy language which functions both at the application layer and at the transport layer. This policy language allows users to define rules that are source destination based. A user is able to specify bandwidth requirements which 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, we have implemented: *allow*, *drop*, *log*, and *copy*. Source and destination use a custom defined port object which can either be 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). We have implemented a highly efficient version of the Entropy Estimation algorithm used by Dorfinger et al [15] Actions can take parameters, such as a destination port for *copy* and *log*.

E. Implementation

We used the OpenFlow v1.3 [25] implementation of Ryu [4] using the Zodiac WX [10] hardware. Building on top of the OverFlow protocol, we used Ryu for our development. Our domain language parser is implemented in textX as it allows for easy implementation of new policy languages. We have also implemented the ability for users to define their own custom action and write their own python function for it.

V. Experimental Results

In this section we experimentally evaluate ROS-Defender. We first describe the experimental setup,

A. Experimental Setup

Turtlebot3 robot: This system consists of a Turtlebot3 robot, a Northbound Zodiac Wx switch, and a laptop which is acting as both the SDN controller and ROS master/decision maker. The data used was a SLAM path traversal of an already mapped area. We also mapped the same area 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 being sandboxed from each other through the use of *cgroups*.

Virtual robot: This system is a purely virtual version of our Turtlebot3 system. It leverages the Gazebo package (a well-known robotics simulator) to emulate the sensor input for a variety of different of robotic systems, and uses *Mininet* for the networking component. This experiment was selected to compare the performance of the virtual system with that of the real world system and to provide a broad base of robots for comparison. We use the SLAM algorithms for all of the 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. We analyze different emulated robots of various complexities in order to demonstrate the viability of ROS-Defender across several robot types:

- Turtlebot - Standard simulated robot for ROS platform[7]
- Husky - An outdoor rugged ground vehicle [2]
- AR Drone - Simulated Autopilot for the AR Drone platform [16]
- ABB industrial robot - Simulated pick and place industrial robot with ROS and Robot Studio [8]
- Udacity - Simulator for a self driving car [9]

B. ROSDN Evaluation

Performance 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 fully update to the new model. 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), as well as 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 design of the robot system 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.

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 I: 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.

ROSDN overhead. We ran our test systems in four separate states, without SROS or ROSDN, with SROS and no ROSDN, with ROSDN and no SROS, and with both ROSDN and SROS. For each run, we examined the average latency for each sensor (i.e. the time from which something happened in the real world to when the robot was made aware of it), the maximum size that a topic reached, and the time between system start and the completion of the simulation. All of our results are summarized in Table I.

ROSDN adds an overhead of 46% to the latency of sensor data for the physical experiment and 33% for the virtual experiment. We were unable 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 actually being sent over a wireless network, instead of 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 rules 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 we choose to analyze an already initialized robot to measure the additional overhead of ROSDN during normal operation. These results can be found in Table II. We found that the overall overhead on the system for completing basic tasks was much smaller then the previously measured message overhead would suggest. As demonstrated in Table II, the overhead of the time it took to complete a task was much smaller once ROSDN reached a stead state. We believe that this is due to the robot's run time being constrained more by motor speed then network processing speed. We find these results to be comparable to those of other security research for ROS[22].

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 II: ROS Virtual Result comparisons

VI. Related work

We overviewed work to secure ROS in Section II. We now describe efforts to using SDN for security and securing SDN. SDN has been proposed to provide a policy-based security architecture for securing the communication in multiple AS domains [27] or mobile apps and devices [18]. In terms of securing SDN, the authors of [11], define a new set of standards for secure SDN systems, while the authors of [20], proposed a secure and dependable SDN control platform in order to address the various threats, including replication, diversity and secure components. Secure ROS (not to be confused with SROS) is a novel project the adds IPSEC to ROS without requiring any alterations to the core API [5].

VII. Conclusion and future work

In this paper we proposed ROS-Defender a, comprehensive security architecture for ROS based robotic systems. ROS-Defender does not require changes to the existing ROS code since it's using available ROS application programming interfaces and SDN level mechanisms to monitor and execute access control actions.

References

[1] Apparmor. https://wiki.ubuntu.com/AppArmor. Accessed: 2018-03-27.

[2] Husky unmanned ground vehicle. https://www.clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/. Accessed: 2018-09-27.

[3] Ros technical overview. http://wiki.ros.org/ROS/TechnicalOverview. Accessed: 2018-03-27.

[4] "ryu software-defined networking framework.". https://osrg.github.io/ryu/. Accessed: 2018-06-02.

[5] Secure ros. http://secure-ros.csl.sri.com/. Accessed: 2019-03-01.

[6] Sros. http://wiki.ros.org/SROS. Accessed: 2018-03-27.

[7] Turtlebot3 waffle. https://www.generationrobots.com/fr/402716-turtlebot-3-waffle.html. Accessed: 2018-08-02.

[8] Using simulated robot in robot studio. http://wiki.ros.org/abb/Tutorials/RobotStudio. Accessed: 2018-09-27.

[9] Were building an open source self-driving car. https://github.com/udacity/self-driving-car. Accessed: 2018-09-27.

[10] "zodiac wx". https://northboundnetworks.com/products/zodiac-wx. Accessed: 2018-06-02.

[11] A. Akhunzada, E. Ahmed, A. Gani, M. K. Khan, M. Imran, and S. Guizani. Securing software defined networks: taxonomy, requirements, and open issues. *IEEE Communications Magazine*, 53(4):36–44, April 2015.

[12] B. Breiling, B. Dieber, and P. Schartner. Secure communication for the robot operating system. pages 1–6, April 2017.

[13] J. Chu. Army robotics in the military. https://insights.sei.cmu.edu/sei_blog/2017/06/army-robotics-in-the-military.html. Accessed: August 02, 2018.

[14] B. Dieber, B. Breiling, S. Taurer, S. Kacianka, S. Rass, and P. Schartner. Security for the robot operating system. *Robot. Auton. Syst.*, 98:192–203, Dec. 2017.

[15] P. Dorfinger, G. Panholzer, and W. John. Entropy estimation for real-time encrypted traffic identification (short paper). In J. Domingo-Pascual, Y. Shavitt, and S. Uhlig, editors, *Traffic Monitoring and Analysis*, pages 164–171, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[16] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart. *Robot Operating System (ROS): The Complete Reference (Volume 1)*, chapter RotorS—A Modular Gazebo MAV Simulator Framework, pages 595–625. Springer International Publishing, Cham, 2016.

[17] A. Giaretta, M. D. Donno, and N. Dragoni. Adding salt to pepper: A structured security assessment over a humanoid robot. *CoRR*, abs/1805.04101, 2018.

[18] S. Hong, R. Baykov, L. Xu, S. Nadimpalli, and G. Gu. Towards sdn-defined programmable BYOD (bring your own device) security. 2016.

[19] S.-Y. Jeong, I.-J. Choi, Y.-J. Kim, Y.-M. Shin, J.-H. Han, G.-H. Jung, and K.-G. Kim. 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, pages 147–148, New York, NY, USA, 2017. ACM.

[20] D. Kreutz, F. M. Ramos, and P. Verissimo. Towards secure and dependable software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 55–60, New York, NY, USA, 2013. ACM.

[21] S. Lagraa, M. Cailac, S. Rivera, F. Beck, and R. State. Real-time attack detection on robot cameras: A self-driving car application. In *IEEE International Conference on Robotic Computing (IRC)*, 2019.

[22] F. Martn, E. Soriano, and J. M. Caas. Quantitative analysis of security in distributed robotic frameworks. *Robotics and Autonomous Systems*, 100:95 – 107, 2018.

[23] R. Rahimi, C. Shao, M. Veeraraghavan, A. Fumagalli, J. Nicho, J. Meyer, S. Edwards, C. Flannigan, and P. Evans. An industrial robotics application with cloud computing and high-speed networking. In *IEEE International Conference on Robotic Computing (IRC)*, pages 44–51, 2017.

[24] S. Rivera, S. Lagraa, and R. State. Rosploit: Cybersecurity tool for ros. In *IEEE International Conference on Robotic Computing (IRC)*, 2019.

[25] The Open Networking Foundation. OpenFlow Switch Specification, Jun. 2012.

[26] R. Toris, C. Shue, and S. Chernova. Message authentication codes for secure remote non-native client connections to ros enabled robots. pages 1–6, April 2014.

[27] V. Varadharajan, K. K. Karmakar, and U. Tupakula. Securing communication in multiple autonomous system domains with software defined networking. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 195–203, 2017.

[28] R. White, H. I. Christensen, and M. Quigley. SROS: securing ROS over the wire, in the graph, and through the kernel. *CoRR*, abs/1611.07060, 2016.