ROS-Immunity: Integrated Approach for the Security of ROS-enabled Robotic Systems

This paper was downloaded from TechRxiv (<https://www.techrxiv.org>).

LICENSE

CC BY 4.0

SUBMISSION DATE / POSTED DATE

28-09-2020 / 29-09-2020

CITATION

Rivera, Sean; Iannillo, Antonio Ken; State, Radu (2020): ROS-Immunity: Integrated Approach for the Security of ROS-enabled Robotic Systems. TechRxiv. Preprint. <https://doi.org/10.36227/techrxiv.13013336.v1>

DOI

[10.36227/techrxiv.13013336.v1](https://doi.org/10.36227/techrxiv.13013336.v1)

ROS-Immunity: Integrated Approach for the Security of ROS-enabled Robotic Systems

Sean Rivera, Antonio Ken Iannillo, Radu State
SnT, University of Luxembourg
firstname.lastname@uni.lu

Abstract

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 Introduction

The Robot Operating System, also known as ROS, is a library framework to develop robotic applications [36]. It is an open-source project based on a modular design and, thus, highly reusable. In the last ten years, ROS became a de-facto standard for industrial applications powered by the ROS-Industrial [40], a consortium of companies all over the world that extends the advanced capabilities of ROS software to industrial relevant hardware and applications.

Robots are characterized by a certain degree of movement and autonomy to perform tasks [20], thus consisting of software that controls both a sensor and a mechanical part acting upon the processed sensor data. 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. There is a need to communicate between robots over a network to perform a common task.

ROS enables the development of robotic systems at a fine-grained scale. It reduces complexities by splitting functionalities into nodes. A node is a process that performs a specific computation. 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. A message is a typed data structure, and an anonymous publish/subscriber transport system [5] is in charge of exchanging them. A node can produce a message by publishing it with a specific label, that is a topic. The system then delivers the message to the nodes that subscribed to that specific topic. By communicating through topics, nodes are unaware of whom they are communicating with. A particular node, namely the master node, is always present in a ROS-enabled system. The master node keeps track of the entire system, maintaining an internal graph of all topics. It is also in charge of hosting the parameter server that acts as a global variable repository of the robotic system.

However, ROS is notorious for the absence of security mechanisms, only partially covered by recent advancements with ROS2 [33], and its lack of robustness when an attacker hijacks arbitrary messages. For example, the ROS master is unable to enforce the mapping between topics and nodes, resulting in severe vulnerabilities [13] [12]. Further, developers usually build their systems by re-using third-party and untrusted nodes available to the open community, providing a significant attack vector.

We propose an integrated solution, *ROS-Immunity*, with small overhead that allows ROS users to harden their systems against attackers, especially those who have access to the exchanged messages between ROS nodes. *ROS-Immunity* is even able to detect on-going attacks that exploit new vulnerabilities in ROS systems. The solution 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 robots of the target system and can detect and block malicious messages or other malicious behavior.

In this paper, we considered four use-cases for ROS systems where multiple ROS nodes are distributed over the network to showcase our solution’s abilities. First, we considered a self-driving car where components are spread throughout the vehicle. Then, we targeted MONA [1], a swarm robotic system where distinct robots collaborate to reach a common goal. Lastly, we considered both a centralized and a real-world decentralized assembly line based on ARGoS [34], where many task optimized robots work in a sequential network.

Our main contribution consists of providing a light security solution for ROS that is capable of covering the whole chain from discovering vulnerabilities to protecting from attacks. We investigate both centralized and decentralized ROS systems. We evaluated *ROS-Immunity* on the three use-cases and demonstrated that we could block all the encoded attacks, with a maximum worst-case false positive rate of 17%. The worst-case overhead of our solution is 70% on the CPU performance, 150% on the memory usage, and 27% for the network performance. However, our typical overhead is 10% for the CPU performance, 25% for the memory usage, and 15% for the network performance. We found that our system performs very efficiently with centralized systems that have a small number of robots, and decentralized systems with a large number of robots. *ROS-Immunity* performs worst with decentralized systems with a small number of robots.

2 ROS System Designs

In this section, we present the 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.

2.1 Centralized System

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

Figure 1: ROS Centralized System Design with a single master node and communication channel from the master node to the other nodes.

the master node and that it is the only ROS master they can communicate to. It is also assumed that robots must communicate with the ROS master for functionality. 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. This implies that every single robot in a connected system can reliably communicate to a centralized system (or robot). 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.

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

This 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 operating 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 [27], 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

Figure 2: ROS Decentralized System Design with multiple master nodes and hierarchical communication model among the robots.

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 tend to 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.

Our goal for centralized systems is to ensure that the master node is always protected and aware of the underlying status of every robot in the system. We aim 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.

2.2 Decentralized System

We define a decentralized system as 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. This means that a decentralized ROS1 system cannot easily apply any cryptographic key-based system. Decentralized systems are most common in swarm robotics and in systems where there is unreliable networking, although other examples may exist.

When dealing with decentralized ROS systems, our 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. This indicates that robots have to deal with

problems of byzantine fault tolerance and that they must rely on consensus algorithms to detect misbehaving robots in their midst. We assume 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, our solution is still valid, however, some targets become far more critical than others. We assume that 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, we do not assume that any part of the system can force the rest into a fail-safe state. However, we assume that if an issue is detected, misbehaving robots can be quarantined, and users can be notified to activate the physical shut-off.

Our goal 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. We aim to provide a solution that a user can use to determine the source of issues for efficient remediation quickly.

3 Overall Architecture

In this section, we will outline the threat model we use to create *ROS-Immunity* and the architecture designed to address the two ROS system designs.

3.1 Threat Model

For our threat model of the ROS system, we begin by assuming 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 [27] [13] [7]. We make no assumptions about the security of any ROS packages or a user’s ability to patch the system’s underlying software.

As far as attackers go, we assume 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, which is 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, we assume 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. We assume 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.

We do not assume any shared secrets or other requirements on the attacker’s behalf, essentially assuming that the attacker has a full white-box of our system and the underlying robots. Finally, we assume 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 [10] 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 our threat model, we do not assume that we can protect ROS Bridge from malicious JSON messages, only that we can prevent the publication of exploits.

Owing to our threat model, we design our solution with the following restrictions.

- As we assume 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 we assume 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 our security approach. Although all communication in a system is encrypted at the start, 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.

3.2 Solution

We propose a novel solution, *ROS-Immunity*, to implement 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 Figure 3: 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 while identifying 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 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 as well as 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 [6]. 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 [24] [42], property-based testing [41], Honeypots [19] [29], and fuzzing [18] [44]. 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 all 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. Static code analysis, meanwhile, 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 1 lists the strengths and weaknesses of these systems when used alone. Any system cannot address the complex security ecosystem surrounding ROS systems.

The need for uniformity further complicates this. Once a developer has chosen which combinations of tools they want to employ, they must use a standard of fingerprinting. Without

Figure 3: *ROS-Immunity*'s three components to implement an integrated security mechanism in ROS system addressing the security gaps in current ROS systems.

Type	Strengths	Weaknesses
Static Code Analysis [24] [42]	Fast and easy to use	Large amounts of false positives
Honeypot [19] [29]	Detects real world vulnerabilities	Requires constant maintenance
Fuzzing [18] [44]	Finds runtime issues not seen by other models	Performance heavy and programmatically difficult
Property Based Testing [41]	Leverages existing test infrastructure	Requires manual intervention to analyze bugs

Table 1: Strengths and weaknesses of known ROS security tools

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. Methods such as these do exist, such as the development of describing robotic vulnerabilities similar to the MITIR CVE [30]. Once a developer attains all fingerprints, a method of both vulnerability analysis and vulnerability mitigation (until a patch has been created) is needed.

For this paper, we utilize a combination of anomaly detection, Honeypots, and fuzzing 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.

In order 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 such that they can 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 there is a majority, or all rules will be denied and assumed to be an attack. While this adds a layer of complication, we demonstrate that this method cannot be compromised, as the distributed firewall will identify and stop attacks.

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

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 rules such as 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 Spe-

cific 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 possible that fully encompasses the vulnerability fingerprint. In this way, we do not accidentally block any packets that are not vulnerabilities and ensure each rule is narrowly-targeted. This automatic rule generation allows a user to maintain a list of fingerprints, or 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 6.

Rules are composed of three key components: behaviors, affected systems, and values. The first, behaviors, encompasses 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 ultimate goal of the automatic rule generator 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. By 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 the publication of this paper, 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 be damaging to 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 the rule applies to. 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 between them 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 [21]. 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 6.

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 [14]. This can lead to patching and vulnerability mitigation issues as each developer must either maintain separate branches of common

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.

We believe that a ROS-aware firewall, in conjunction with an adequate ROS cryptography system, 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. Our firewall system is developed to block unauthorized access to topics and service connections and to filter out any messages with identified vulnerabilities until the system is patched.

In theory, a ROS firewall would need to be aware of both ROS middleware syntax and underlying network connections. 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 the proper functioning of any robotic system, as small changes can significantly impact such systems. Theoretically, this may be conducted using access control enforcement and filtering suspected compromised packets, but, to the best of our knowledge, 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 harder than topics and masters running on the same system. Users would need to ensure that the robot correctly asked the master node for the location of the topic 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, we will outline our implementation for a distributed ROS firewall that addresses all of these issues, for both centralized and decentralized robotic architectures. Our solution provides an efficient way to take novel vulnerabilities and translate them into patches, as well as to detect when a robot has been compromised and mitigate damage to other robots in the network.

Our 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 individual code in kernel space [17]. XDP is

an extension to eBPF, allowing users to bypass the traditional Linux network stack for efficient packet processing [2]. We choose to use Tendermint [26] as our Byzantine fault-tolerant protocol. It supports both centralized and decentralized ROS systems and has a simple interface for alerting users of potential vulnerabilities. Our centralized solution includes one keystone master firewall that all robots share, and the decentralized solution requires each robot to maintain its master firewall.

In Figure 4, we outline 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 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, we utilize the byzantine fault-tolerant Tendermint protocol where robots can propose new rules, and other robots can vote based on a distributed verification system. This verification system works as follows: a robot can individually verify a proposed rule if they have the 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 our 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 dynamically adapt to an attacker: 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

Figure 4: Integrated Firewall Architecture for centralized (on the left) and decentralized (on the right) ROS system designs.

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 a majority 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 of when they last updated their rules to ensure all robots 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 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 underlying 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 5, allowing for developers to rapidly patch known vulnerabilities without changing the underlying code.

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 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.

7 Use Cases & Experimental Evaluation

To test our system, we designed four separate common ROS use-cases which we believed covered the most common real-world robotic situations. For each use-case, we analyze nine configurations of nodes and rules. We vary both robot count and rule count to examine the cost of *ROS-Immunity*. We simulated each use-case using Docker [31] 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.

Our first use-case is a ROS self-driving car with a cluster of specialized nodes communicating over a TCP network. This car is modeled after the Autoware [22] [23] 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, as well as a back-up control node in case the first fails. All iterations of this experiment are considered a centralized system.

Our next use-case is based on an example from swarm robotics. We 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 their 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.

Our final use-case is a ROS factory with robots working along an assembly line. For this use-case, we evaluated two separate architectures: 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, we test robots 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 arms. 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.

We chose to split this use-case by type of system as they are both common infrastructures with unique issues, allowing a direct comparison of our 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 design of the decentralized system: 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, we confirmed our setup had similar behavior to a real-world ROS installation.

Use Cases	Separate Proces- sors	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 2: Breakdown of our use-cases and experimental evaluation

To evaluate the use-cases, we constructed bag files with a randomized mixture of regular data and added attacks. We looked at both attacks that had firewall rules as well as attacks that did not. For attacks that were known, we ensured that the firewall successfully stopped them in all cases and then measured the overhead and false-positive rate. 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.

We conducted multiple trials 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, we compared three configurations of robots. For each configuration of robots, we considered three configurations of rules: a count of 10, 20, or 50. We then repeated each trial 16 times to get a better representation of how our system performs. In total, we performed 576 experiments.

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 our 12 use-case and robot configurations, we injected vulnerabilities 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, we selected a random grouping of vulnerabilities and placed them randomly throughout the bag-file.

Simply, our 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. We detect false-positives by exploring discrepancies in the network data.

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 3: False positive rate per trial

The results of our experimentation are summarized in Table 3, as well as Figures 5, 6 7, and 8. Overall, the performance of *ROS-Immunity* was highly promising, producing small processor and memory overhead in all use-cases. We discovered that the decentralized implementation scaled well with the number of robots, while the centralized was far more efficient with smaller systems. The firewall in-line filtering stopped 100% of known attacks with a worst-case 17% false-positive rate.

Table 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 false-positive as a false-positive.

To calculate our false-positive rate for the upper bound, we considered the worst-case scenario. For each trial, we combined the three different experimental setups with a different number of robots with the same number of rules. We identified any scenarios where a packet was filtered. We assumed that if any packet not added by us as a vulnerability was filtered from the system, the entire trial had a false-positive in at least one of the robots. 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 we are dealing with safety-critical systems wherein filtering out vital information from the robot when it is not part of an attack is an unacceptable risk. We report 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 expected result. While normally 10% is considered a high false-positive rate, in this case, we consider absolute worst-cases, and the actual false-positive rate is likely much lower. We believe that there is an opportunity for future improvement of the false-positive rate through 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 results of the self-driving car 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 is added for both control and experimental to load the vulnerability into the experimental bag-file correctly. The addition of new data is conducted in both control and experimental for consistency, but no vulnerabilities are 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 experiment runs, the network shape is non-deterministic, and each reconstruction requires a large amount of bandwidth usage. This leads to high variance in the network usage between runs

Figure 5: Car

Figure 6: Swarm

and many inconsistent results.

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

The results of the centralized factory 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% ($\sigma = 6\%$) for 10 robots, 11% ($\sigma = 3\%$) for 20 robots, and 6% ($\sigma = 1\%$) for 50 robots. It has an average memory overhead of 45% ($\sigma = 4\%$) for 10 robots, 16% ($\sigma = 3\%$) for 20 robots and 8% ($\sigma = 4\%$) for 50 robots. It has an average network overhead of 3% ($\sigma = 2\%$) for 10 robots, 5% ($\sigma = 1\%$) for 20 robots, and 15% ($\sigma = 2\%$) 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 centralized 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, we tested 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. We found that the car

Figure 7: Centralized Factory

Figure 8: Decentralized Factory

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. We also found that 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 was still able to 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 9 we calculated the per second power requirements of *ROS-Immunity* versus our control system using the ARM embedded power formula found in Mao *et al.* [28]¹. We normalized the power draw of each robot in the system. We found that *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 [39].

Figure 9: Normalized Power Overhead

8 Related work

This section will briefly present ROS security research and those that work close to the use cases presented in this paper, even if not based on ROS.

Dieber *et al.* [11] presented RosPenTO, a semi-automated

¹We chose to exclude GPU power from the calculations

tool for testing ROS. The tool injects fake messages into the ROS middleware, demonstrating the lack of security in ROS. Indeed, an attacker can easily impersonate any subscriber node and querying the master for sensitive information.

Santos *et al.* [41] targeted ROS systems for a property-based testing framework. The framework automatically generates test scripts for various configurations of a ROS system, while detecting for crashes or violation of defined properties.

Bihlmaier *et al.* [3, 4] designed a framework to monitor ROS systems at run-time to find configuration errors and bottlenecks, namely ARNI. ARNI collects and presents data about the messages exchanged by the ROS node, providing metrics such as CPU and memory usage. The authors also proposed a detection mechanism to warn the user about the erroneous state in the system.

Another proposed monitoring system is Drums by Monajemi *et al.* [32]. Drums communicate with ROS middleware, unveiling the system-level interactions of the nodes, the middleware, OS, and the robot(s) environment. Using this information, Drums is mainly used for debugging purposes.

Rivera *et al.* [38] proposed a testing tool for ROS, namely ROSploit. It can be used for modeling and exploiting vulnerabilities in ROS. ROS-Defender [37] was an attempt to integrate security for ROS, including a security event management system, an intrusion prevention system, and a firewall using SDN. While ROS-Defender was able to protect centralized ROS1 systems, it was not able to monitor ROS2 systems nor decentralized ROS1 systems. Additionally, ROS-Defender was found to have too large of an overhead for many ROS applications.

Koscher *et al.* [25] presented composite attacks in automobile systems by infiltrating in the electronic control unit. They highlighted the lack of detection or enforced protection mechanisms of essential services such as diagnostic and reflashing. Furthermore, they stressed the importance of securing the vehicle bus where third-parties components are automatically trusted, leaving space to attackers.

Checkoway *et al.* [9] also targeted modern automotive systems and demonstrated the feasibility of remote exploitation through several attack vectors (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.

Quarta *et al.* [35] analyzed the reference architecture and real-world industrial robotic systems to assess their security capabilities. They formalized security challenges to address in the short-, medium-, and long-term. Among them, attack detection and system hardening are listed as counter-measures 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.

Ferrer [15] 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.* [16] They 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.* [43] also exploited blockchain technology and decentralized programs (smart contracts) to secure coordination in swarm systems and to detect Byzantine robot.

Similarly, Cameron *et al.* [8] exploited blockchains to develop a multi-chain system to detect Byzantine actors in swarm robotics systems.

9 Conclusion and future work

In this paper, we outlined *ROS-Immunity*: a novel integrated security solution for ROS. We demonstrated that *ROS-Immunity* can vastly improve the security of several different robotic systems with low overhead. We implemented it in 4 use-cases exemplifying typical ROS systems: a self-driving car, a swarm, a centralized assembly line, and a decentralized one. We performed several experiments and measured both the overhead and power consumption of *ROS-Immunity* and the ROS systems. *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, .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. We believe 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 on the overhead.

References

- [1] F. Arvin, J. Espinosa, B. Bird, A. West, S. Watson, and B. Lennox. Mona: an affordable open-source mobile robot for education and research. *Journal of Intelligent & Robotic Systems*, 94(3-4):761–775, 2019.
- [2] G. Bertin. Xdp in practice: integrating xdp into our ddos mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, volume 2, 2017.
- [3] A. Bihlmaier, M. Hadlich, and H. Wörn. *Advanced ROS Network Introspection (ARNI)*, pages 651–670. Springer International Publishing, Cham, 2016.

- [4] A. Bihlmaier and H. Wörn. Increasing ros reliability and safety through advanced introspection capabilities. In E. Plödereder, L. Grunske, E. Schneider, and D. Ull, editors, *Informatik 2014*, pages 1319–1326, Bonn, 2014. Gesellschaft für Informatik e.V.
- [5] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, 1987.
- [6] T. Bonaci, J. Herron, T. Yusuf, J. Yan, T. Kohno, and H. J. Chizeck. To make a robot secure: An experimental analysis of cyber security threats against teleoperated surgical robots. *arXiv preprint arXiv:1504.04339*, 2015.
- [7] G. Caiazza, R. White, and A. Cortesi. Enhancing security in ros. In *Advanced Computing and Systems for Security*, pages 3–15. Springer, 2019.
- [8] A. Cameron, M. Payne, and B. Prela. Research and implementation of multiple blockchain byzantine secure consensus protocols for robot swarms. 2018.
- [9] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*, volume 4, pages 447–462. San Francisco, 2011.
- [10] C. Crick, G. Jay, S. Osentoski, B. Pitzer, and O. C. Jenkins. Rosbridge: Ros for non-ros users. In *Robotics Research*, pages 493–504. Springer, 2017.
- [11] B. Dieber, B. Breiling, S. Taurer, S. Kacianka, S. Rass, and P. Schartner. Security for the robot operating system. *Robotics and Autonomous Systems*, 98:192–203, 2017.
- [12] B. Dieber, S. Kacianka, S. Rass, and P. Schartner. Application-level security for ros-based applications. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4477–4482. IEEE, 2016.
- [13] B. Dieber, R. White, S. Taurer, B. Breiling, G. Caiazza, H. Christensen, and A. Cortesi. Penetration testing ros. In *Robot Operating System (ROS)*, pages 183–225. Springer, 2020.
- [14] P. Estefo, J. Simmonds, R. Robbes, and J. Fabry. The robot operating system: Package reuse and community dynamics. *Journal of Systems and Software*, 151:226–242, 2019.
- [15] E. C. Ferrer. The blockchain: a new framework for robotic swarm systems. In *Proceedings of the future technologies conference*, pages 1037–1058. Springer, 2018.
- [16] E. C. Ferrer, T. Hardjono, M. Dorigo, and A. Pentland. Secure and secret cooperation of robotic swarms by using merkle trees. *arXiv preprint arXiv:1904.09266*, 2019.
- [17] M. Fleming. A thorough introduction to ebpf. *Linux Weekly News*, 2017.
- [18] J. C. Han and Z. Q. Zhou. Metamorphic fuzz testing of autonomous vehicles.
- [19] C. Irvine, D. Formby, S. Litchfield, and R. Beyah. Honeybot: A honeypot for robotic systems. *Proceedings of the IEEE*, 106(1):61–70, 2017.
- [20] Robots and robotic devices — Vocabulary. Standard, International Organization for Standardization, Geneva, CH, Mar. 2012.
- [21] K. Iwanicki, M. Van Steen, and S. Voulgaris. Gossip-based clock synchronization for large decentralized systems. In *IEEE International Workshop on Self-Managed Networks, Systems, and Services*, pages 28–42. Springer, 2006.
- [22] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, 2015.
- [23] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPs)*, pages 287–296. IEEE, 2018.
- [24] J. Kim, J. M. Smereka, C. Cheung, S. Nepal, and M. Grobler. Security and performance considerations in ros 2: A balancing act. *arXiv preprint arXiv:1809.09566*, 2018.
- [25] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, et al. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462. IEEE, 2010.
- [26] J. Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 1(11), 2014.
- [27] G. Lacava, A. Marotta, F. Martinelli, A. Saracino, A. La Marra, E. Gil-Uriarte, and V. M. Vilches. Current research issues on cyber security in robotics. 2020.
- [28] Y. Mao. Detailed power measurement with arm embedded boards. 2018.

- [29] J. McClean, C. Stull, C. Farrar, and D. Mascarenas. A preliminary cyber-physical security assessment of the robot operating system (ros). In *Unmanned Systems Technology XV*, volume 8741, page 874110. International Society for Optics and Photonics, 2013.
- [30] P. Mell, K. Scarfone, and S. Romanosky. A complete guide to the common vulnerability scoring system version 2.0. In *Published by FIRST-forum of incident response and security teams*, volume 1, page 23, 2007.
- [31] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.
- [32] V. Monajjemi, J. Wawerla, and R. Vaughan. Drums: A middleware-aware distributed robot monitoring system. In *2014 Canadian Conference on Computer and Robot Vision*, pages 211–218, 2014.
- [33] Open Source Robotics Foundation, Inc. ROS 2 Design. <http://design.ros2.org/>. [Online; accessed 1-June-2020].
- [34] C. Pinciroli, V. Trianni, R. O’Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, et al. Argos: a modular, multi-engine simulator for heterogeneous swarm robotics. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5027–5034. IEEE, 2011.
- [35] D. Quarta, M. Pogliani, M. Polino, F. Maggi, A. M. Zanchettin, and S. Zanero. An experimental security analysis of an industrial robot controller. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 268–286, 2017.
- [36] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.
- [37] S. Rivera, S. Lagraa, C. Nita-Rotaru, S. Becker, and R. State. Ros-defender: Sdn-based security policy enforcement for robotic applications. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 114–119. IEEE, 2019.
- [38] S. Rivera, S. Lagraa, and R. State. Rosploit: Cybersecurity tool for ros. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 415–416. IEEE, 2019.
- [39] F. J. Rodríguez-Lera, V. Matellán-Olivera, J. Balsa-Comerón, Á. M. Guerrero-Higueras, and C. Fernández-Llamas. Message encryption in robot operating system: Collateral effects of hardening mobile robots. *Frontiers in ICT*, 5:2, 2018.
- [40] ROS-Industrial Consortium. ROS-Industrial. <https://rosindustrial.org/>. [Online; accessed 1-June-2020].
- [41] A. Santos, A. Cunha, and N. 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*, pages 56–62, 2018.
- [42] A. Santos, A. Cunha, N. Macedo, and C. Lourenço. A framework for quality assessment of ros repositories. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4491–4496. IEEE, 2016.
- [43] V. Strobel, E. Castelló Ferrer, and M. 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*, pages 541–549. International Foundation for Autonomous Agents and Multiagent Systems, 2018.
- [44] V. M. Vilches, L. A. Kirschgens, A. B. Calvo, A. H. Cordero, R. I. Pisón, D. M. Vilches, A. M. Rosas, G. O. Mendia, L. U. S. Juan, I. Z. Ugarte, et al. Introducing the robot security framework (rsf), a standardized methodology to perform security assessments in robotics. *arXiv preprint arXiv:1806.04042*, 2018.