# Smart Malware that Uses Leaked Control Data of Robotic Applications: The Case of Raven-II Surgical Robots

Keywhan Chung[1], Xiao Li[1], Peicheng Tang[2], Zeran Zhu[1], Zbigniew T. Kalbarczyk[1], Ravishankar K. Iyer[1], and Thenkurussi Kesavadas[1]

[1]University of Illinois at Urbana-Champaign
[2]Rose-Hulman Institute of Technology

## Abstract

In this paper, we demonstrate a new type of threat that leverages machine learning techniques to maximize its impact. We use the Raven-II surgical robot and its haptic feedback rendering algorithm as an application. We exploit ROS vulnerabilities and implement smart self-learning malware that can track the movements of the robot's arms and trigger the attack payload when the robot is in a critical stage of a (hypothetical) surgical procedure. By keeping the learning procedure internal to the malicious node that runs outside the physical components of the robotic application, an adversary can hide most of the malicious activities from security monitors that might be deployed in the system. Also, if an attack payload mimics an accidental failure, it is likely that the system administrator will fail to identify the malicious intention and will treat the attack as an accidental failure. After demonstrating the security threats, we devise methods (i.e., a safety engine) to protect the robotic system against the identified risk.

## 1 Introduction

A number of attempts have been made to leverage machine learning (ML) techniques to realize malicious intentions. For instance, adversarial learning was used to effectively deceive data-driven models by strategically injecting malicious input [8, 25, 45, 47] to identify the target of an attack [28], or to infer information hidden behind encrypted data [26, 33]. In this context, smart malware that employs ML techniques represents a new concept for the implementation of sophisticated attack strategies. Such malware can infer attack strategies based on live operational data and trigger an attack at the most opportune time so as to maximize the impact.

For threats that use self-learning malware, robotic applications turn out to be a fascinating target. Like other cyber-physical systems (CPSes), robotic applications incorporate sensors and actuators that are connected through a network that passes around data. Their (i) *relatively weak security* [5, 13, 31], (ii) *abundance of data that can be used to infer actionable intelligence* [4, 9,54], and (iii) *close proximity to and direct interactions with humans* (such that a successful attack could have a life-threatening impact) [14, 22, 23] make robotic applications a tempting target for advanced threats.

To demonstrate the feasibility of smart malware, we have built an injection module as a prototype. Our prototype smart malware eavesdrops on the communication between the robot components of a near-real-time system (as an input for the smart malware), uses the leaked data to infer intelligence on when to trigger the payload (or take control over the robot), and executes the payload at the most opportune time (i.e., the output of our smart malware) so that it can maximize the impact. While our attack model applies to any robotic system, in this paper, we use the Raven-II surgical robot [3] and its haptic feedback rendering algorithm as a target application.

Raven-II is driven by the Robot Operating System (ROS) [44], an open-source framework that has been widely deployed across various robotic applications (i.e., more than 125 applications [38]), and its resiliency is critical to varying domains (e.g., robotic surgery, aviation, and manufacturing). However, the most commonly used ROS contains vulnerabilities [15] that leak data (e.g., robot state) transmitted within the application, and those data can become the basis from which smart malware can *learn* about the system behavior and use this information to decide when to trigger an attack. We exploited ROS vulnerabilities and implemented *smart malware* that tracks the movement of the robot's arms and triggers the attack payload when the robot is in a critical state of a (hypothetical) surgical procedure. After demonstrating the security threats, we discuss the methods (i.e., a safety module) that we devised to protect the robotic system against the identified risk.

What makes our malware stealthy is the invisibility of its learning process to security monitoring systems. Unlike common malware, which is installed *in* a victim system, our malware runs outside the physical components of the robotic application. The ROS allows any new node/process to register with a master (core) node; hence, an attacker can register its malicious node to the robotic application without being noticed. By keeping the learning procedure internal to the

malicious node, an adversary can hide all malicious activities (except for its network activities with genuine nodes of the target application) from security monitors that might be deployed. Hence, only the impact of the attack, which mimics accidental failures, is observable to the system administrator. As a result, it is likely that malicious faults will be seen as accidental failures (especially if the network traffic is not being monitored). Note that additional network traffic introduced by our malware prototype is negligible (about 0.24% of the volume of the genuine traffic).

The contributions of this paper are the following:

- We show the possibility of a *real attack on a surgical robot* that exploits known vulnerabilities in the underlying runtime environment, ROS. The vulnerabilities allow a malicious entity to operate as a man-in-the-middle (MITM), with the ability to eavesdrop (i.e., leak robot control data) and overwrite communication among the robot components (i.e., effectively take control of the robot).
- We demonstrate *smart malware logic* that can infer the most opportune time of attack from the information obtained through exploitation of the vulnerabilities in ROS. Our experiment with three use cases that mimic (hypothetical) surgical operations of different levels of complexity shows that the ML algorithm (DBSCAN) used by our malware can determine the position of the robot end-effector with respect to the target object and use this information to trigger the execution of the payload. Specifically, the DBSCAN algorithm triggers the injection of the attack payload (i.e., corruption of data used to control the robot) when the robot arm is in close proximity to the target object (i.e., there is less than 10 mm distance between the robot end-effector and the target object).
- We present a set of *unique faults* that, when used as an attack payload, can threaten the integrity of the surgical operation. The faults consist of realistic scenarios that can be disguised as accidental failures, such as network packet drop, data corruption, or bugs in the control software.
- We implement a *ROS-generic safety module* that can detect abnormalities introduced by attacks and can bring the robot to a safe state. Specifically, the safety module detects the shutdown signal generated by the *roscore* in the case of a name conflict (i.e., a new node registers itself with an already existing name). If that happens, the safety module terminates the new node, takes over the control of the robot, and returns it to a predefined safe state to prevent further impact.

While we demonstrate the feasibility of the advanced threat in the context of Raven-II and its underlying framework (i.e., ROS), the design of the smart malware is sufficiently generic that it can be used on other robotic systems that generate a stream of sensor data from input sensors and robot control data to the physical robots. As summarized in [1], an attacker can intrude into a robotic system through various entry points (e.g., third-party networks, vulnerable workstations, and vulnerable or incorrectly configured firewalls or gateways). Once smart malware has established an MITM attack (i.e., it can listen to and overwrite control data), its smart injection module can infer the critical time of the operation of any robot. The attack payloads (i.e., the faults to be injected), on the other hand, are specific to the robotic application.

## 2 Motivation for smart malware

Machine learning techniques have been applied in different domains (e.g., image processing and natural language processing) to derive intelligence from data. Researchers and engineers in cyber security have also deployed ML-based techniques as part of an effort to advance methods for detecting malicious activities. However, not much work has considered the possibility that adversaries could take advantage of machine learning algorithms to devise attack strategies. More specifically, a few studies have investigated the potential impact of attacks that are supported by machine learning algorithms [40, 41]. In this paper, we define smart malware as malicious software that can, by itself, derive intelligence from data obtained from the victim system.

Smart malware is available only at a cost (i.e., high computation workload). However, we find reasons that might justify the overhead: access to rich data and an ability to achieve high impact with minimized remote interaction between the malware (software) and the attacker (human). (That is, to a certain extent, machine learning algorithms can replace human-driven analysis in designing/customizing malware.) Notably, long and unusual remote connections often lead to exposure of attackers. Furthermore, the computational load imposed by the execution of the smart malware can be obfuscated with techniques such as the "low and slow" approach, whereby attackers intentionally reduce the computation workload despite having to tolerate a longer time of execution.

For machine-learning-driven threats, cyber-physical systems (especially robotic applications) turn out to be tempting targets. In cyber-physical systems, sensors and monitors are deployed across the system to gather information (e.g., on images, sounds, temperature, and flows). Data collected from input sensors are sent to controllers or computation units that derive control variables or decisions, which are passed to the actuator to update the state of the system. While traditional robots were contained within a single physical system, the new concept of distributed robotics (or collaborative robotics) is expanding the boundary of robotic systems. (E.g., with remote surgery, a physician can perform surgery from a remote location.) A key enabler for this new mode of attacking the system is a protocol for sharing data across a network. However, if the protocol is not properly designed for security, it can introduce vulnerabilities that eventually exploited by smart malware.

For instance, a publish-subscribe model is a common messaging pattern in which the information is shared between

the publisher and the subscriber. Its advantages include scalability and loose coupling between the publishers and the subscribers. However, such advantages introduce side effects that impact the security of the system. Without authentication and encryption, unauthorized entities can read messages and leak data. Such data become a baseline for learning, from which malicious entities can derive actionable intelligence. In this paper, we demonstrate the threat by using the Raven-II surgical robot (running on top of ROS) as the target for such an attack strategy.

## 3 Background: Robots, ROS, and Raven-II

Robots have been adopted across different application domains. For example, in manufacturing, robot manipulators assist human workers; drones are deployed in agriculture, entertainment, and military operations; and surgical robots support surgeons in performing medical procedures. For such applications, robots play a critical role. A robot's failure to make a correct and timely movement can lead to catastrophic consequences, such as injuring people near the robots in factories or risking a patient's life during surgery. This study focuses on the resiliency of a surgical robot against malicious attacks. We use the Raven-II surgical robot [3] and its haptic rendering algorithm as an application to demonstrate the security threat, and suggest methods to cope with the risk.

**Robot Operating System (ROS).** The Robot Operating System (ROS) is an open-source framework for programming robots [44], and is commonly used by various robotic applications. According to its official website, ROS is widely deployed across more than 125 different robots, including mobile robots, drones, manipulators, and humanoids [38]. The framework is being developed to support collaborative development by experts from different domains (e.g., computer vision or motion planning) and provides hardware abstraction, device drivers, libraries, and a communication interface [44]. For instance, the OpenCV library [6] provides interfaces that can be used to add vision to robotics applications, and the OpenNI library [35] focuses on integrating 3D sensors into robots. As ROS provides the core underlying runtime environment, the security of ROS is critical in ensuring the correct operation of the robot.

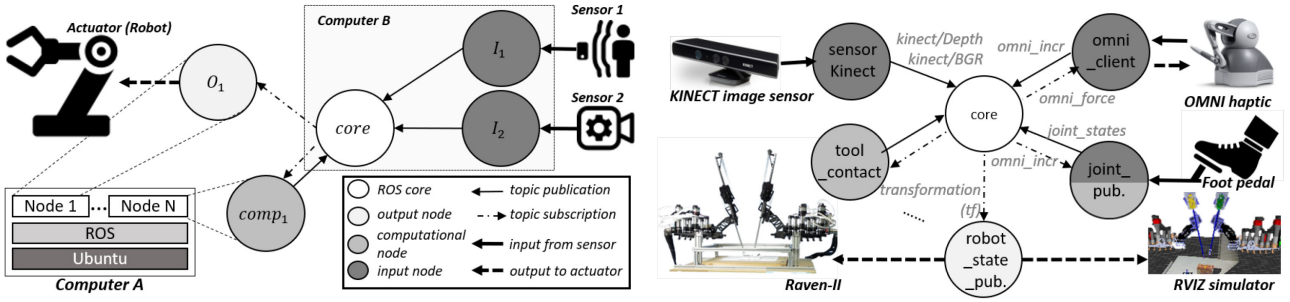
As shown in Fig. 1a, a ROS-based application consists of multiple *ROS nodes*. They can be running on a single physical machine or can be distributed across multiple machines (i.e., *Computers A* and *B* in the figure), as long as they share the *ROS core*, which is deployed on the computer declared as the *ROS master*. Each node communicates over the network, and the ROS core serves as the central server for all nodes. Data are exchanged in the context of a *topic*, where a *topic* is a data structure defined to deliver a specific context type; e.g., the image sensor data are exchanged in the form of multidimensional arrays, which consist of the RGB color codes for all pixels captured by the image sensor. A node, either a *publisher* or a *subscriber*, registers itself to the ROS core for the topic that the node is about to publish (or subscribe to). The ROS core then passes the information (i.e., the IP address) of the publisher to the subscriber waiting for the topic, so that the subscriber can establish a TCP connection with the publisher. After a handshaking protocol and transmission of the metadata that include the structure of the topic message, the two entities start passing the message by using a ROS-specific protocol.

The ROS nodes can be classified into three types: input nodes, output nodes, and computational nodes. An *input node* is a node connected to a piece of hardware (e.g., an image sensor or a haptic device) that provides input to the robot application. The input node, using the device driver provided by (or interfaced with) ROS, collects the data and converts the data into a ROS message as defined for the topic. Once the message is ready, the input node publishes it. The input node should have declared itself to the ROS core as a publisher for a topic. A *computational node* is a node that takes the input data to produce the output (e.g., a command to a robot actuator); it subscribes from the input node and publishes to the output node. Finally, an *output node* is a node connected to an actuator (i.e., the robot). The values (i.e., robot states and control commands) subscribed from the computational node are converted into a form that the hardware can interpret, and passed to the hardware. The output data determine the (joint) state of the robot.

The ROS framework comes with the *RVIZ* software package, which allows users to test and visualize the operation of the robot applications in a virtualized environment. *RVIZ* takes the physical specifications of the robot and displays the mesh of the robot; it is heavily used to test robot designs without using a physical robot.

**Raven-II and haptic force feedback rendering engine.** In this paper, we study the resiliency of a ROS application in the context of a surgical robot (i.e., Raven-II) and its haptic feedback rendering engine. Leveraging the open-architecture surgical robot, the authors of [30] present a hardware-in-the-loop simulator for training surgeons in telerobotic surgery. The simulator, in addition to having all the features of the Raven surgical robot, introduces a novel algorithm to provide haptic feedback to the operator and, hence, offer a touch sensation to surgeons. Unfortunately, commercially available surgical systems[1] (e.g., da Vinci by Intuitive [52]) do not provide haptic feedback to the operator. The traditional approach for haptic feedback uses physical force sensors to determine the force applied to the robot. Since the instruments (on which the force sensors are installed) are disposable, that approach turns out to be costly. Instead, the authors of [30] proposed an indirect haptic feedback rendering approach that does not rely

[1] In 2017, the FDA approved a new surgical system [49] with haptic force feedback. However, we do not have sufficient information to understand the underlying technology and, hence, the capability of the system.




(a) Generic robot running on ROS.

(b) Raven-II with its haptic feedback rendering algorithm.

Figure 1: Software architecture of robotic applications.

on force sensor measurement, but instead uses image sensor data to derive the force feedback.

In our study, the haptic feedback rendering algorithm, as implemented in the augmented Raven-II simulator, utilizes information from a depth map (a matrix of distances from the image sensor to each pixel of a hard surface) to derive the distance from the object (e.g., a patient's tissues) to the robot arm. Using the current position of the arm and the measured distance to the object, the algorithm returns an interactive force value that generates resistance in the haptic device.

Figure 1b provides an overview of the software architecture of Raven-II and its haptic feedback rendering algorithm. The haptic algorithm takes input from the Kinect image sensor and the OMNI haptic device to control the Raven-II robot (or its virtual representation in *RVIZ*). The `sensorkinect` node parses the image data (as BGR and depth) from the Kinect image sensor, packages the data into ROS messages, and publishes the messages to the ROS core as topics (*kinect/BGR* and *kinect/Depth*). The `omni_client` node is connected to the OMNI haptic device for user input. The `omni_client` node shares the processed operator input as a topic (*omni_incr*). A set of nodes, dedicated to running the algorithm, subscribe to the topics from the ROS core and derive the force feedback, which the `omni_client` sends to the haptic device.

The *kinect/Depth* topic from `sensorkinect` is used to derive the distance from the robot arm to the object. However, in deriving the distance, the algorithm needs a reference frame. It leverages the ArUco library [21, 46], which is an Open-Source library commonly used for camera pose estimation. With the ArUco marker (i.e., a squared marker) location fixed and used as a reference point, we can derive the location of the robot arm(s) relative to the marker. Using that information, the algorithm can derive the distance from the robot arm to the object by using (i) the transformation from the marker to the robot, (ii) the transformation from the image sensor to the marker, and (iii) the transformation from the image sensor to the object. Because the transformation from the robot arm to the object is evaluated in near-real-time, the algorithm can provide timely haptic force feedback to the OMNI device.

## 4 Approach

Cyber security is often referred to as a "cat and mouse" game. In this paper, we are considering potential advances in cyber threats, assuming that adversaries will eventually take advantage of machine learning techniques (if they are not already doing so). In this section, we present our approach for corrupting a robotic application with self-learning malware. We developed this approach to raise awareness of the potential threats, and to promote preparation for responding to this threat. The methodologies included in our approach can be used for (i) preemptive identification of vulnerabilities, (ii) hardening of robotic applications against potential threats, and (iii) design of detection/mitigation methods.

**Threat model.** In our threat model, we assume:

- The attacker *can penetrate into the control network of the robot*. As presented in [13], ROS applications are often connected to a public network without proper protection. One can provide a level of protection by virtually isolating the control network (i.e., by deploying a VLAN). However, it would be possible to intrude into the virtual network either with stolen credentials or by exploiting a weak link (i.e., a vulnerable computer that has access to the VLAN). In the context of attacks on surgical robots, a survey on potential entry points of a hospital network can be found in [1].
- The attacker *understands the operation of ROS*, and has access to ROS-provided APIs (which are easily obtainable online). With remote access to the ROS master, one can execute ROS commands.
- The *target robot runs on top of ROS 1*. Our attack model is designed for ROS 1 (e.g., *Kinetic* and *Melodic*), which is still the most commonly deployed version despite the release of ROS 2 (discussed in detail in Section 7.1) in 2015. Software patches have been issued to fix the vulnerabilities in ROS, but, as we discuss in Section 7.1, the patches merely require attackers to take another step to neutralize them. Hence, in describing our attack model, we assume the default setting of the most commonly used ROS.

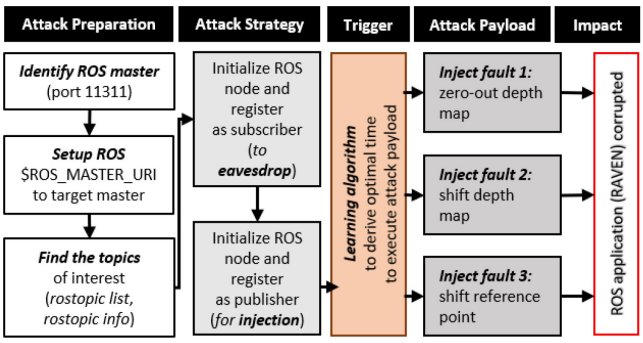



Figure 2: Approach overview, from attack preparation to impact to ROS application.

**Approach overview.** Vulnerabilities present in the ROS framework allow unauthorized entities to eavesdrop on messages passed across ROS nodes. Utilizing the obtained data, a malicious entity can identify an ideal time to trigger an attack and corrupt the operation of the robot by injecting faulty input or output commands. In Figure 2, we present an overview of our approach. During the **attack preparation** phase, we identify the victim (i.e., a ROS master that is remotely accessible) and its critical components. Once they are identified, the **attack strategy** can be applied to perform a man-in-the-middle (MITM) attack. As the malware eavesdrops on the sensor and control data of the robot, the smart malware runs a learning algorithm to infer the location of the target object. When the object is identified (i.e., the algorithm returns a cluster) and the robot reaches the predicted location of the target object, a **trigger** is raised to initiate the **attack payload**. In the following, we describe the details of our approach.

## 4.1 Attack preparation

To deploy the attack, the first step is to identify machines that are running ROS as a master (core) node. Using a network scanning tool, we scan for the default port for ROS masters (i.e., 11311, a well-known port for ROS masters) [13]. Once the master and its IP address are known, we set up ROS on our machine (which mimics a remote attacker) and update the ROS master's Uniform Resource Identifier (URI) variable to that of the identified master. Using the ROS APIs, we search for the topics of interest (i.e., the topics registered to the ROS master are used as a signature for identifying the ROS application).

## 4.2 Attack strategy: ROS-specific MITM

In corrupting a ROS application, we take advantage of the vulnerabilities in ROS and execute a ROS-specific man-in-the-middle attack. As described in Section 3, ROS provides a set of interfaces (*publish*/*subscribe*) that ROS nodes can use to communicate; the ROS core serves as the arbitrator. While the communication might include sensitive data, the ROS 1 framework does not provide options for authenticating or validating the ROS entities. (I.e., any ROS node that can access the master can register itself as a publisher to write messages or as a subscriber to read messages.) After configuring the ROS setup to connect to the victim ROS master (attack preparation; see Section 4.1), our malware can initiate a subscriber that eavesdrops on the network communications. To take control of the robot, it kicks out a genuine node and publishes malicious data while masquerading as the original publisher. Without noticing the change in the publisher, the robotic application takes the malicious data as an input (or command) and updates the state of the robot accordingly.

## 4.3 Trigger: Inference of critical time to initiate the malicious payload

Most security attacks are detected when the attack payload is executed [50]. Once detected, the attacker (or the malware that the attacker had installed) is removed from the system. Consequently, in many cases, the attacker may have one chance to execute the payload before being detected. As a result, it is realistic to consider the case in which an attacker tries to identify the ideal time to execute the attack payload (in our case, to inject a fault) in order to maximize the chances of success. A common approach is to embed a trigger function into the malware, which checks for a condition and executes the payload only when the condition is satisfied.

In [2], Alemzadeh et al. presented an attack model that is triggered by a prediction of the robot state derived by a side-channel attack. In the model, the attacker installs malware on the robot control system, eavesdrops on a USB packet, and infers the state of the robot (i.e., either "engaged" when the surgeon's input is updating the position of the robot, or "disengaged" when the position of the robot is not being updated). The robot state (which is controlled by pedal input from the surgeon) is an effective indicator in determining when malicious input would be fed into the robot. However, in such an approach, it is hard to accurately determine the time window during which the robot is performing critical activities; e.g., it is more critical when the robot is cutting tissue than when it is transitioning towards the target object.

In this study, we present an approach that leverages a well-studied learning technique to infer the *critical time* to trigger the attack payload, so as to maximize the impact.

**Inference of object location.** During a surgical operation, the robot usually moves within a limited range defined by the nature of the surgical procedure. Hence, the precision in identifying 'the time when the robot is touching (or maneuvering close to) the target object' can help in triggering the attack at the most opportune time so as to maximize the impact. For instance, when the robot is moving from its idle location to the patient on the operating table, the robot is operating in an open space without obstacles. Hence, visual input is sufficient

to allow the surgeon to operate. Furthermore, the surgeon will not even notice whether the haptic feedback rendering algorithm is operational, as there is no surface that the robot would touch (i.e., there is zero force feedback). On the other hand, when the robot is inside the abdomen of the patient, it is operating in limited space packed with obstacles (e.g., organs) and with blind spots that the image sensor cannot monitor. In that situation, correct operation of the rendering algorithm is critical. Also, the shorter the distance from the robot (at the point of the trigger) to the target object, the less time it takes the surgeon to respond[2] upon discovering the failure of the rendering algorithm (which can be determined only by noticing the lack of force feedback when a surface is touched). In this paper, *we analyze the spatial density of the robot end-effector position* throughout the operation *to infer a time* when the robot (i.e., the surgical instrument) is near the object.

**Algorithm.** We use unsupervised machine learning to determine the location of the target object with respect to the position of the robot's end-effector(s). Specifically, we adopted the density-based spatial-clustering algorithm with noise (DBSCAN) [20, 48] to accomplish this task. The DBSCAN algorithm takes two parameters, ε and *numMinPonts*. The maximum distance parameter (ε) defines the maximum distance between neighboring data points. Iterating over all data points, the algorithm checks for neighbors whose distance from a data point is less than ε. If the number of neighbors is less than *numMinPoints*, the data point is considered noise (i.e., the data point is not part of a cluster). Otherwise, the algorithm checks whether the neighbors form a cluster. Any clusters already formed by a neighbor are merged into the current cluster. Although the DBSCAN algorithm is known for its sensitivity to the choice of parameters, the attacker does not have much information from which to derive the right parameters. Based on the data subscription frequency (i.e., 80 Hz for our eavesdropper) and our conservative assumption that at least 10% of the overall operation time corresponds to the critical procedures[3] (i.e., 10 seconds for our data from ∼ 100 seconds of robot operation), we set *numMinPonts* to 800 (i.e., $subscription\_frequency \times seconds\_of\_stay$). Also, we consider points (corresponding to the robot's end-effector position) within 1 cm of each other to be "close," and define ε=10. With a goal of demonstrating the feasibility of self-learning malware (not of presenting the best algorithm or parameters for a clustering problem), we find the parameter pair (ε, *numMinPoints*) to be accurate enough for our study. The optimization of the parameters is outside the scope of this paper.

[2]Similar to the concept of braking distance when driving a car.

[3]From a set of medical studies, we find that the mean of the total operation time was 178.2 minutes [12] and that the mean time for a critical procedure was 22.2 minutes [19] (i.e., 12.4% of the mean procedure time).

## 4.4 Attack payload: Fault injection

While the attack strategy in Section 4.2 is generic to the ROS framework, the payload is specific to the ROS application under study (i.e., Raven-II and its haptic feedback rendering algorithm). As part of assessing the resiliency of the haptic feedback rendering engine, we designed a set of faults that can be injected on-the-fly. The faults were designed through a careful study of Raven-II's operation and its rendering algorithm. In this paper, we present three fault models that cause the haptic feedback rendering engine to fail to prevent the operator from penetrating the surface of the object under operation. The three fault models are representative in mimicking realistic cases of (i) loss of information during transmission of data, (ii) data corruption, (iii) a glitch in sensors, and/or (iv) a bug in the software algorithm. None of the faults are specific to the environment (i.e., the faults are not affected by custom settings of the robot in a certain environment). Hence, understanding of the application (without needing to understand certain custom configurations) is sufficient for designing effective faults.

**Fault 1: Loss of granularity in the depth map.** As discussed in Section 3, the haptic feedback rendering algorithm relies heavily on image sensor data. Our first fault model demonstrates a case in which the quality of the image from the sensor is degraded. More specifically, we consider a case in which the granularity of the depth map has become sparse due to (i) hardware problems in the image sensor or (ii) loss of data during transmission of the depth data. In this fault model, we randomly choose a certain percentage of the pixels, for which we neutralize the depth information (i.e., set it to zero, which can be interpreted as setting the distance to that of the ground surface). By carefully choosing the rate at which pixels are dropped, we can disguise an attack as natural noise, despite its critical impact.

**Fault 2: Shifted depth map.** The second fault model considers a case in which an entity with malicious intent manipulates a ROS message to obfuscate the visual data provided to the operator. Just as we dropped the depth information from the image sensor in Fault 1, we can overwrite the depth map message with shifted values, causing the rendering algorithm to provide incorrect haptic feedback that the operator will rely on. In our experiment, we shifted the depth map of the object under operation by 50 pixels to the right. As the ROS message that contained the BGR information remained untouched, the 3D rendered image of the object was incomplete. Similarly, an attacker can shift the data in the BGR message and deliver the malicious visual image to the Raven operator.

**Fault 3: Corrupted reference frame.** As part of rendering haptic feedback, the application needs to derive the distance from the object (under operation) to the robot arms, as the location of the object can change on every run or during the operation of the robot. An attacker can corrupt the coordinates of the reference frame and make the rendered feedback

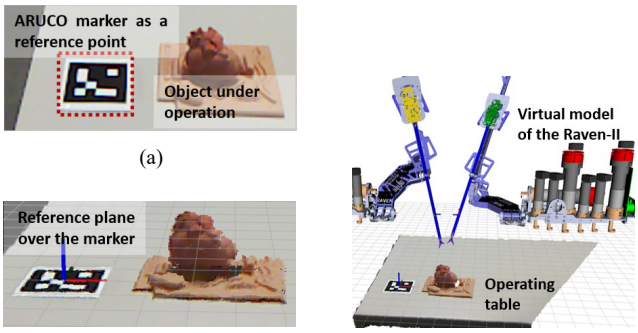



Figure 3: Experimental setup with (a) BGR image of the operating table; (b) *RVIZ* representation of the operating table in 3D; and (c) *RVIZ* representation of the operating room, including Raven-II.

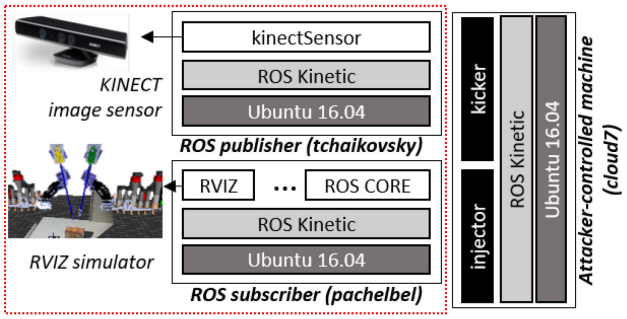



Figure 4: Overview of the system setup for the experiment.

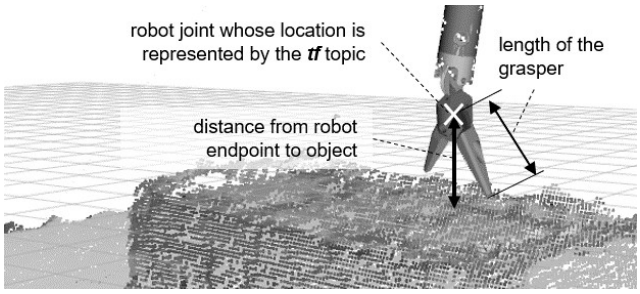



Figure 5: Evaluation of the distance from the robot arm to the object.

become invalid. In this fault model we modify the reference frame during the transmission of the coordinates from the image sensor node to the computational node. Note that the ArUco detection-based reference frame is applicable in experimental settings such as ours. In commercial surgical robots, the known position of trocars (i.e., pen-shaped instruments used to create an opening into the body [7, 29]) are used as the reference frame, and the fault model would need to be modified accordingly.

## 5 Experiment design

**Experimental setup.** In order to mimic the settings of a surgical operation, we set up a mock-up of a heart (the object under operation in Figure 3a) on an operating table. We placed an ArUco marker to calibrate the configuration of the object (i.e., the mock-up of the heart). Once the image sensor passed the information to the Raven-II simulator, the operating table and the target object were rendered in 3D, as shown in Figure 3b. Note that the 3D axes were added to the image upon the algorithm's detection of the marker. Using the predetermined transformation from the marker to the robot, RVIZ can place the virtual representation of the robot over the operating table (Figure 3c). For the demonstration of the smart malware, the mock-up heart was replaced with a simpler shape, i.e., a cuboid (see Figures 5 and 10–12). However, the performance measurements were not affected by the simplification of the shape of the object.

**Systems setup.** To demonstrate the smart malware, as depicted in Figure 4, we set up three Linux (Ubuntu 16.04) machines running ROS (i.e., Kinetic, one of the most recent versions of ROS). While the nodes for Raven-II and the rendering algorithm can be distributed across any combination of machines, we simplified the configuration by distributing the application across two machines (*tchaikovsky* and *pachelbel*, shown in Figure 4). The third machine (*cloud7*) is owned by the attacker. The two machines, both running Raven-II, reside in the same (virtual) network (marked with the dotted box in Figure 4), whereas *cloud7* resides in the same network only when executing the attack strategy (see Section 4.2). The rendering algorithm is designed to run with the (physical) Raven-II robot. However, we limited the experiment to a simulated environment to protect the physical robot from potential damage. Although the experiment was limited to a simulated environment, the faults and their impact still apply to the physical robot.

**Data.** With the MITM established (as described in Section 4), we were able to eavesdrop on all messages transmitted between the ROS nodes. Such messages included ones communicating the robot's joints' state (i.e., the angle of each joint), which determined the robot's end-effector position. We took advantage of the ROS-provided API (i.e., `TransformListener()` [37]) to derive the position of each joint. Using the API, we (in the shoes of an attacker) could collect data on the x, y, and z coordinates of the robot end-effector in three-dimensional space. We considered three scenarios to mimic surgical operations of different levels of complexity: (i) a surgeon focuses on a single region of the target object; (ii) a surgeon operates on two regions of the target object; and (iii) a surgeon operates on three regions of the target object. Note that the movements of the robot arms (manipulated by the surgeon) follow a trajectory specific to a given surgical procedure. Because of our lack of real data on such trajectories, we imitated rather complex routes to challenge our algorithm.

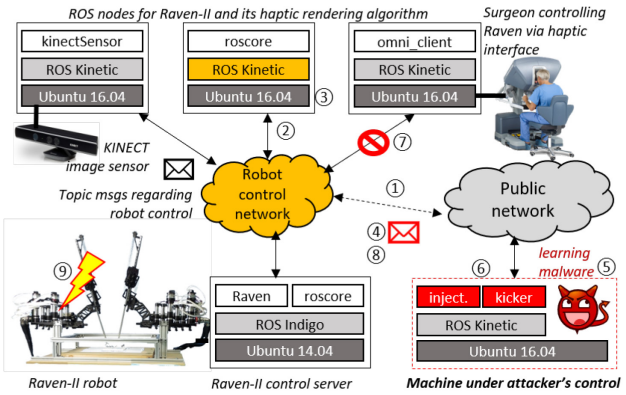



Figure 6: Overview of the steps taken by the malware.

**Evaluation metrics.** The goal of an attacker is to trigger the execution of the attack payload at the most opportune time so as to maximize the damage, e.g., hurt a patient or damage the robot. In order to achieve that objective, the attacker must precisely determine when the robot is operating near (if not in contact with) the target object. The clustering algorithm (in Section 4) indirectly derives the decision by monitoring the trace density of the robot arm in 3D space. To evaluate the effectiveness of the decision, we measured the distance from the robot arm to the object (see Figure 5). That would not be not possible for a real attacker, as the location of the object would remain unknown. We evaluate the predictions by using a threshold (i.e., 10 mm) that defines "close to object." Applying the definition, we derive the number of predicted instances that would lead to a successful attack. (We consider an attack to be "successful" if the execution is triggered when the distance from the robot arm to the object is less than the threshold).

**Automated malware execution.** In Figure 6, we show how an attack using our smart malware would proceed. The attacker starts by getting access to the control network of the robot (❶). This step could be accomplished by scanning for the target ROS application connected to the public network [13], or stealing the credentials of a legitimate user in the control network (via social engineering or phishing attacks). With access to the control network, the attacker scans the network for the 11311 port to find the ROS master (❷). In ❸ the attacker checks the version of ROSand disables all patches that remediate the vulnerabilities of ROS 1. Next, the attacker can deploy the smart self-learning malware. First, the malware subscribes (❹) to topics of interest (e.g., tf, a ROS-generic topic for the x, y, z coordinates of the robot end-point). By running the DBSCAN algorithm, the malware can label each point (i.e., member of a cluster or noise)(❺). When the robot arm position is classified as a cluster, the malware triggers the payload execution (❻), which registers the malicious publisher with the name of a genuine publisher. Because of the name conflict, in ❼ the ROS master shuts down the genuine publisher (i.e., `omni_client`) and the malicious topic (i.e., faulty data) is passed to the ROS application (❽). Because of the faulty data, the operation of Raven is corrupted, which puts the patient at risk (❾).

## 6 Results

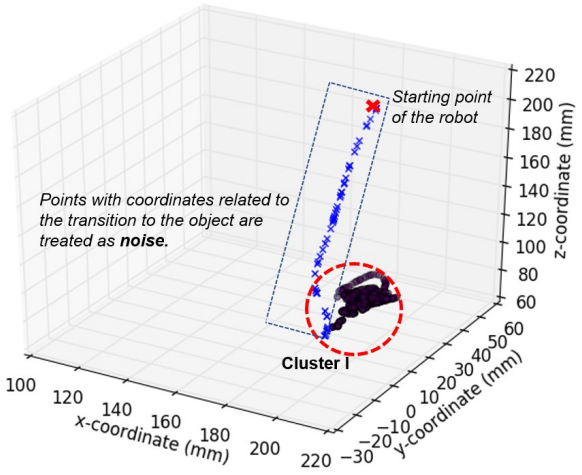
In this section, we present our results from inferring the time to trigger the attack payload and injecting realistic faults.

### 6.1 Determining attack triggers

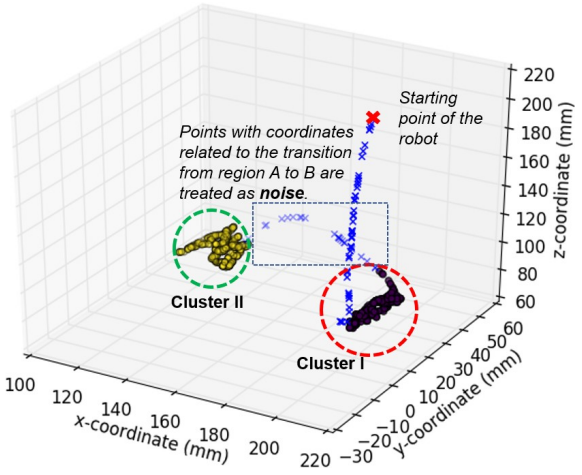
In this section, we evaluate our accuracy in determining the robot's end-effector position with respect to the target object. In Figure 7, we present the results of the clustering algorithm (based on DBSCAN) for the three scenarios: (i) a surgeon operates on a single region of the target object; (ii) a surgeon operates on two regions of the target object; and (iii) a surgeon operates on three regions of the target object. Note that in Figures 7a–7c, an "x" indicates that the point is considered noise, and a circle indicates that the point belongs to a cluster. (Different colors are used to differentiate clusters.) Also, in Figure 5, we compare the clustering results with those from a pedal-detection-based approach (*pedal*) [2].

**Case 1: Single region of operation.** Figure 7a depicts the trajectory of the robot arm for the case in which a surgeon is operating at a single region of the object. The algorithm effectively identifies the data points that correspond to the region of operation and successfully filters out the data points related to the transition of the robot arm from the starting point of the robot arm to the region of operation. In Figure 8a, we present a cumulative distribution of the distance from the clustered points (robot's joint positions) to the target object. While all points of the DBSCAN-derived clusters had a distance of less than 1 cm from the target object, the *pedal-detection-based algorithm* included points related to transition of the robot, which resulted in reduction of the probability of a successful attack. Also, as depicted in Figure 9, the algorithm effectively clusters the instances in which the robot is closer to the object ("clustered1" in Figure 9), as opposed to the points that were labeled as transitions to the object ("transition1" in Figure 9). For our algorithm, the distance (from the robot arm to the object) varied from 0.0 mm to 7.5 mm, and our clustering algorithm was able to filter out the points that corresponded to transitions from the starting point of the robot arm to operational regions (*cluster I* in Figure 7a). The pedal-detection-based approach includes the starting point of the robot arm as a potential trigger for an attack. (Note that the starting point is 121 mm from the object.). As shown in Figure 8a, 99.9% of the DBSCAN-predicted triggers were within 7.1 mm of the object. However, for the pedal detection-based approach, only 80.3% of the predicted points were within 7.1 mm.

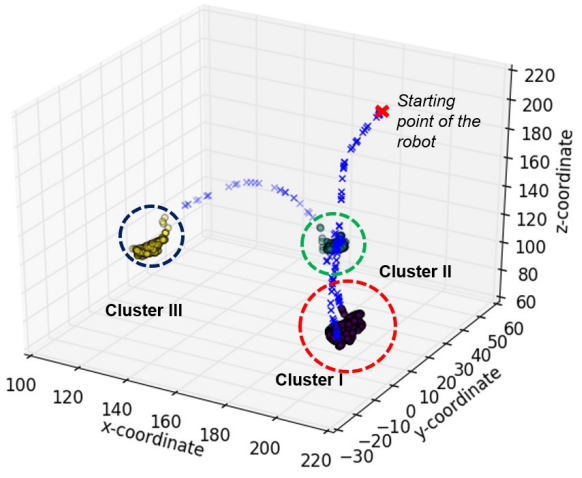
**Case 2: Two regions of operation.** As shown in Figure 7b, the algorithm successfully captured the two regions despite




(a) Single region of operation.

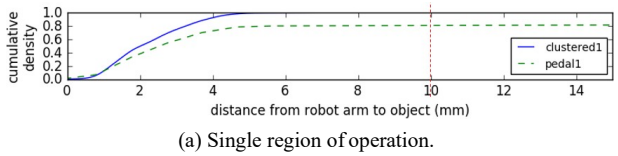



(b) Two regions of operation.

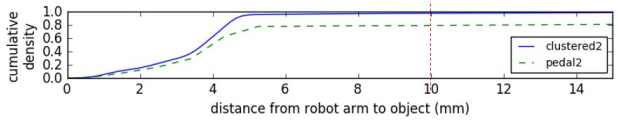



(c) Three regions of operation.

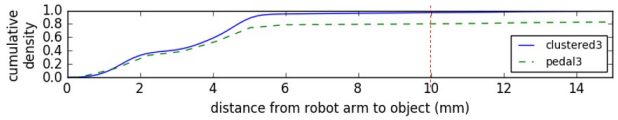
Figure 7: Results of tracing of the robot's arm movements in three hypothetical surgical procedures.




(a) Single region of operation.




(b) Two regions of operation.




(c) Three regions of operation.

Figure 8: Cumulative distribution of the distance from the robot arm to the object.

the complexity added to the operation. While we have added an intermediate transition between the regions of operation, the algorithm successfully filtered out such transitions, and distinguished the two regions. As depicted in Figure 9 and Figure 8b, the clustering algorithm was able to find a subset that contained the majority of the points that were closest to the object (i.e., 0.26 mm $\leq$ *distance* $\leq$ 20.44 mm).

**Case 3: Three regions of operation.** In Figure 7c, we present the case in which the surgery takes place in three adjacent regions. The algorithm successfully detected all three regions. Also, all points clustered by our algorithm turned out to be within 19.8 mm of the object (i.e., all points in the clusters had a *distance*$\leq$19.82 mm). For the pedal-detection-based algorithm, 19.2% of the points triggered unsuccessful attacks while with our DBSCAN-based approach, 3.23% would have been unsuccessful.

**Discussion.** Triggering when the instrument is in close proximity to the target object is essential to increasing the likelihood of success. As demonstrated in the experiments, our DBSCAN-based approach effectively predicts points that are close to the target object. As discussed in [53], the success of the DBSCAN algorithm is sensitive to the choice of the two parameters (i.e., $\varepsilon$ and $n$). In this paper, we have taken a trial-and-error approach, which would not be feasible for an attacker with limited information. (I.e., the attacker cannot confirm whether the resulting cluster truly represents the region of interest.) Instead, the attacker can tune the learning algorithm (i.e., find the optimal parameters, $\varepsilon$ and $n$) offline and install the malware with the parameters embedded.

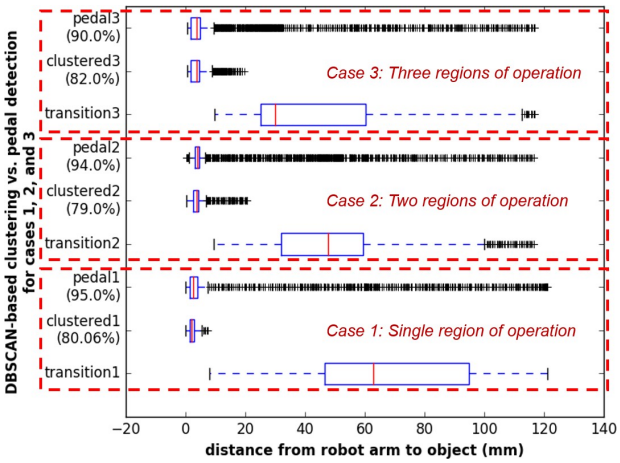



Figure 9: Distribution of the distances from the predicted clusters (predicted either by DBSCAN or by pedal detection) to the object. The labels "clustered" and "transition" indicate coordinates that were predicted as "surgical operation" and "transition of the robot", while "pedal" is for the coordinates filtered by the approach in [2].

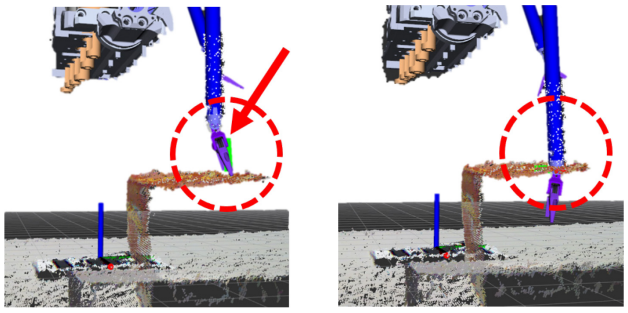
## 6.2 Impact of attacks on the Raven-II haptic feedback rendering algorithm

This section presents the impacts of executing the three attack payloads: (i) loss of granularity in the depth map, (ii) shifted depth map, and (iii) corrupted reference point.

In Figure 10, we present the result of dropping 90% of the pixels from the depth map. (Note that to maximize the visibility of the fault's impact, we have chosen an extreme case and neutralized an unrealistically large portion of pixels.) As a result, the robot arm tip penetrated the surface of the object (Figure 10b), whereas the algorithm should have blocked it from doing so (as seen in Figure 10a). In reality, incorrect rendering of the force feedback can damage or endanger the underlying surface and make the robot suffer a heavy load.

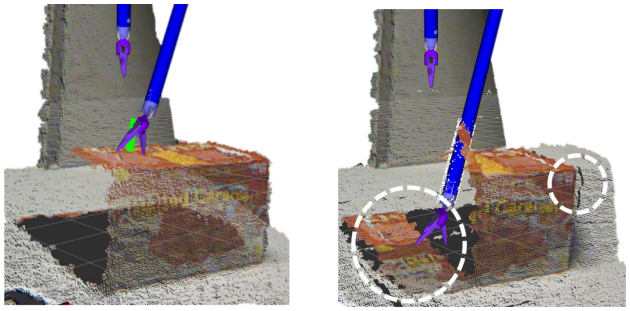
In Figure 11, we show the impact of shifting depth map information during transmission of the information from the publisher to the subscriber. The figure shows that because of the shifted distance measure, the 3D rendering of the left half of the box is flattened (and indeed would be hard to differentiate from the surface, were it not for the colors), and the original surface on the right side of the box has gained volume. As shown in the figure, this fault model can lead to penetration of the object by the robot that would have been prevented by the non-corrupted image.

The last fault we studied was corruption of the derived reference point (the ArUco marker). The object in Figure 12a has the reference point set on the ArUco marker, whereas the reference point in Figure 12b has been shifted upward. The distance between the object and the robot arm (marked with a double-headed arrow) has been updated accordingly. (I.e., the distance in (b) is larger than in (a), while (a) depicts the actual setup of the robot and the object.) For the corrupted distance, the rendered feedback is no longer valid. In the experiment shown in Figure 12, the haptic device did not receive haptic



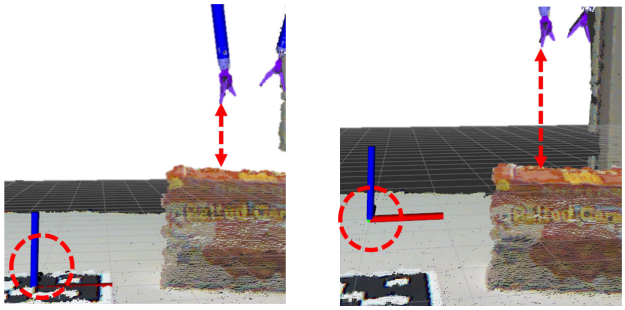
(a) (b)

Figure 10: Simulated Raven operation with (a) uncorrupted depth map and (b) corrupted depth map. Note the difference between the dotted circles.



(a) (b)

Figure 11: Simulated Raven operation with (a) uncorrupted depth map, and (b) shifted depth map. Note the problems inside the contents of dotted circles.



(a) (b)

Figure 12: Simulated Raven operation with (a) uncorrupted camMsg, and (b) shifted camMsg. Note the difference between the reference points (inside dotted circles).

feedback upon touching the surface of the object, and the robot penetrated the object.

## 7 Discussion

**Generalization.** As demonstrated in this paper, the ROS 1 is vulnerable to variations of MITM attacks. We show that our prototype smart malware can utilize the leaked data to trigger an attack at the most opportune time so as to maximize the attack's impact. The inference of the opportune time range for execution of the attack payload reduces the chances of exposure, which helps the malware disguise the attack as an accidental failure. (Recall that the faults were designed to represent accidental failures.) Without the smart triggering, frequent and likely unsuccessful injections of the fault could make the system administrator aware of the malicious intent behind the sequence of failures. To make things worse, our DBSCAN-based approach does not require extensive prior knowledge of the target robotic application. Alemzadeh et al. [2] also introduce a triggering algorithm (i.e., a side-channel attack that predicts the state of the robot from the byte values of specific packets), but we find that our approach is more intuitive and effective[4]. (I.e., our approach simply searches for a dense region (corresponding to a high density of critical activities), and that does not require background knowledge on the design/implementation details of the target robot.) However, despite the smartness of the malware, our attack is limited in its payload. Unlike the fault in [2], which tackles the time gap between a safety check of the input and its execution (i.e., faults are injected after the safety check), our faults (or faulty scenarios) are injected through corruption of the raw data, and that corruption might be detected by safety checks (if such checks were implemented as part of the robotic application).

While we demonstrated the feasibility of a smart malware attack in the context of the Raven-II surgical robot and its haptic feedback rendering algorithm, our threat model exploits vulnerabilities in the underlying framework (ROS). Hence, a robotic application running on the most common version of ROS, ROS 1, is vulnerable to MITM attacks and to smart malware that exploits the leaked data. Furthermore, the generic idea of malware driven by ML algorithms can be expanded to any computing infrastructure that generates a stream of data if the data contain actionable intelligence that smart malware can infer and the system has vulnerabilities that allow malicious entities to access data. While we leverage vulnerabilities in the ROS, as discussed in [1], various entry points exist through which malicious entities could intrude into robotic applications. By leveraging such vulnerabilities, our design for smart malware can be revised to target robotic applications in general.

[4]Please note that the goals of the two approaches were different. The goal of the approach in [2] is to infer the time when the corrupted data will be passed to the robot and update its state.

The work performed in this study is not intended to support hackers, but to proactively assess the resiliency of robotic applications, identify vulnerabilities in their design, and drive development of methods to harden robotic systems. For instance, the sensitivity of the haptic feedback rendering algorithm to its input data (as identified during our experiment) requires hardening of the data validation process. That hardening can be done by validating the publisher (to maintain the integrity of the data) or by deploying redundancy in the sensors.

### 7.1 Protection

The leakage of control data being transmitted between components of a robot can lead to inference of sensitive information that can threaten the operation of the robot. As a result, it is critical to secure the robot by (i) assuring that only authorized entities can control robot operation, (ii) securing the communications between the components of the application, and (iii) closely monitoring the robot for anomalies. In this section, we discuss technologies that can be used to secure the application, and their limitations. Also, we introduce our safety module, which detects abnormal circumstances and brings the robot to a predefined *safe state*.

In terms of computer security, MITM attacks have been well-studied, and a number of protection and detection methods have been introduced [11, 24, 32]. For instance, to prevent ARP poisoning (which is a critical step in performing an attack), each machine can have its ARP table set to be static, to prevent unknown entities from updating the table. Also, authentication of the nodes can prevent unauthorized entities from hijacking a session. (I.e., unauthorized ROS nodes should not be able to register to the ROS core, and entries that can publish/subscribe a topic should be defined.)

**Security enhancements for ROS (SROS).** To better secure the communications within the robotics applications, SROS [39, 55] provides TLS support in the socket-level transport. However, the current distribution of SROS is limited to TCPROS (not UDPROS) and robotic applications written in Python (not C++ or Java). As an alternative, one can add a layer of authentication by running all ROS nodes within a VPN (which would require authentication), and that approach is already common. However, it is not rare for malicious entities to intrude into a protected network by using weak or stolen credentials.

**Secure ROS.** Unlike the well-studied TCP MITM attack, our ROS-specific attack model has not been well-investigated. Fortunately, a "fork" of the core ROS packages was released to enable secure communication in the ROS applications [51]. The "Secure ROS" introduced a new configuration file, which specifies the configuration of the application. Furthermore, by utilizing IPSec, Secure ROS ensures that the IP packets cannot be tampered with or spoofed. While Secure ROS enhances the security of ROS applications, the neutralization of the patch

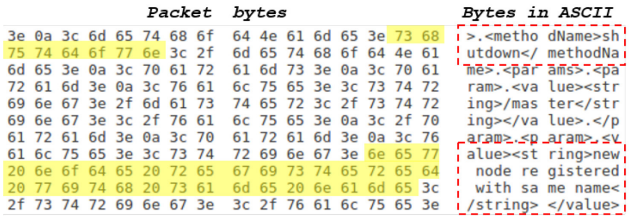



Figure 13: Packet capture of the network packet carrying the "shutdown" command from the roscore, triggered by a conflict in node names.

is not particularly difficult. As the package is an addition to an already existing ROS installation in the system, it has a single parameter (defined in a bash script) that enables the patch. As a result, an attacker can overwrite the parameter to disable the entire security patch, thus returning the system back to the vulnerable ROS.

**ROS 2.** ROS 2 (released in 2015) [36] is a major update from the original ROS. On top of introducing new features to address challenges that arose from extended usage of the framework (e.g., with real-time systems, or groups of robots), the upgrade also covers the vulnerabilities discussed in this paper. The ROS 2 uses the data distribution service (DDS) [34] for publish-subscribe transport. The security enhancements provided by the Secure ROS and SROS patch can be embedded in ROS 2 using DDS. Hence, the vulnerabilities exploited in this paper can be eliminated by using the upgraded framework. However, ROS 2 is not backward-compatible. As a result, the existing applications must be rewritten to take advantage of the new features in ROS 2, and such rewriting is not trivial. Even if the developer manages to re-program the robotic application with the new interface, configuring of the DDS to support the security needs is left to the robot's programmers. Such configuration requires thorough understanding of encryption, certification, and access control. Without security in mind, if the programmer decides not to enable the security features of DDS (e.g., using the eProsima Fast RTPS middleware [18]) or if the programmer makes a mistake in configuring the DDS [16], the vulnerabilities discussed in this paper remain current. As discussed in [27], the DDS and its implementation have limitations (e.g., lack of forward security) and vulnerabilities (e.g., skipping of variable initialization) that attackers can exploit.

## 7.2 Safety module

The limitations of existing technologies mean that the threats described in this paper (especially that specific to ROS applications) remain current and, hence, can occur in any of the 125+ existing robots. As a result, we find a need for a safety module that can (i) detect abnormalities in the robot and (ii) take control of the robot and bring it to a safe state under such circumstances. As discussed in Section 4, when the ROS master detects a name conflict (due to a new node's registering of itself with a name already in use), it terminates the original node and registers the new node with the name in conflict. Our safety module detects the *shutdown* signal transmitted over the network (see Figure 13). Upon detecting the shutdown signal due to a name conflict (see Figure 13), our safety module (operating as another ROS node) terminates the node that publishes the state of the robot joints. By terminating the node, the safety module can prevent the malicious entity from taking control of the robot. Furthermore, if needed, the safety module can bring the robot to a predefined safe state (in our experiment, the reset position of the robot). As the safety module runs on the ROS master with privilege, we can be assured that the shutdown packet will be detected, with minimal risk that the attacker can corrupt the detection. Also, as the detection relies on an unusual signature, we can minimize false positives. (I.e., name conflict is a rare event and when there are multiple nodes with identical names by design, the programmer enables the *anonymous* mode, which pads a hash to the name to avoid the conflict.)

# 8 Related work

**Attacks with learning features.** In [41], an open-source hacking AI, DeepHack, was presented. Powered with a neural network, the tool learns how to intrude into web applications. DeepLocker [28], on the other hand, takes advantage of a deep neural network for target detection. Until the malware detects the target, the malware disguises itself as benign software. Furthermore, the malware encrypts the payload to conceal the malicious intent, which makes reverse-engineering challenging. In [10], the authors leverage a learning technique to infer an attack payload from CPS operational data. The malware in [10] predicts failure-causing abnormalities in the CPS operational data, and injects abnormalities into the control data to corrupt the operation of the CPS.

**Attacks against ROS.** Some vulnerabilities of ROS were discussed in [42, 43]. Using the STOP surveillance system, the authors demonstrate an attack that changes the route of the patrol robot. They proposed the use of IPSec, which was indeed incorporated in the upgrade to ROS 2. The whitelist method proposed by Dóczi et al. [17] has also become part of the new ROS framework. Similarly, [15] discusses a method for preventing malicious publishers and subscribers from interfering with a given ROS node network. The authors ensured broadcast encryption by whitelisting nodes in an authentication server and by requiring any new publisher to run an authentication to certify itself as a legitimate new publisher. Despite the efforts to secure ROS applications, as demonstrated in [13], a significant number of ROS applications that are connected to networks are vulnerable. The authors of [13], by scanning over the whole IPV4 address space, identified more than 100 hosts running as ROS masters. Also, in [5], the authors demonstrate that ROS applications can be vulnerable

to attacks that modify the instructions of a Raven operator (e.g., by manipulating packets to cause loss, reordering, or delay of commands) and to session-hijacking attacks.

## 9 Conclusions

In this paper, we studied the impact of security attacks that exploit security vulnerabilities in ROS to attack robotic applications. More specifically, we demonstrated (i) the possibility of neutralizing the force feedback engine in Raven-II by corrupting a message passed across ROS nodes over the network, and (ii) the possibility of misleading the robot operator by providing incorrect feedback. Our study of ROS and observations on the impact of security attacks reveal a need for advanced security APIs to be provided by the framework. We suggest that the applications be secured in the implementation phase, and be enforced by the framework.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. 18-16673 and 15-45069. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## References

[1] Homa Alemzadeh. *Data-driven Resiliency Assessment of Medical Cyber-physical Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 2016.

[2] Homa Alemzadeh, Daniel Chen, Xiao Li, Thenkurussi Kesavadas, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Targeted Attacks on Teleoperated Surgical Robots: Dynamic Model-based Detection and Mitigation. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pages 395–406, 2016.

[3] Applied Dexterity Website. Accessed May 1, 2019. http://applieddexterity.com.

[4] Rachad Atat, Lingjia Liu, Jinsong Wu, Guangyu Li, Chunxuan Ye, and Yang Yi. Big Data Meet Cyber-Physical Systems: A Panoramic Survey. *CoRR*, abs/1810.12399, 2018.

[5] Tamara Bonaci, Jeffrey Herron, Tariq Yusuf, Junjie Yan, Tadayoshi Kohno, and Howard Jay Chizeck. To Make a Robot Secure: An Experimental Analysis of Cyber Security Threats against Teleoperated Surgical Robots. *arXiv preprint arXiv:1504.04339*, 2015. https://arxiv.org/abs/1504.04339.

[6] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, Inc., 2008.

[7] Charles Chang, Zoe Steinberg, Anup Shah, and Mohan S. Gundeti. Patient Positioning and Port Placement for Robot-assisted Surgery. *Journal of Endourology*, 28(6):631–638, 2014.

[8] Yizheng Chen, Yacin Nadji, Athanasios Kountouras, Fabian Monrose, Roberto Perdisci, Manos Antonakakis, and Nikolaos Vasiloglou. Practical Attacks against Graph-based Clustering. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1125–1142. ACM, 2017.

[9] Xiuzhen Cheng, Yunchuan Sun, Antonio Jara, Houbing Song, and Yingjie Tian. Big Data and Knowledge Extraction for Cyber-Physical Systems. *International Journal of Distributed Sensor Networks*, 11(9):231527, 2015.

[10] Keywhan Chung, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Availability Attacks on Computing Systems through Alteration of Environmental Control: Smart Malware Approach. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 1–12. ACM, 2019.

[11] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. A Survey of Man in the Middle Attacks. *IEEE Communications Surveys & Tutorials*, 18(3):2027–2051, 2016.

[12] Altair da Silva Costa Jr. Assessment of Operative Times of Multiple Surgical Specialties in a Public University Hospital. *Einstein (São Paulo)*, 15(2):200–205, 2017.

[13] Nicholas DeMarinis, Stefanie Tellex, Vasileios Kemerlis, George Konidaris, and Rodrigo Fonseca. Scanning the Internet for ROS: A View of Security in Robotics Research. *arXiv preprint arXiv:1808.03322*, 2018. https://arxiv.org/abs/1808.03322.

[14] Department of Homeland Security. Cyber Physical Systems Security, Feb. 2019. https://www.dhs.gov/science-and-technology/csd-cpssec.

[15] Bernhard Dieber, Severin Kacianka, Stefan Rass, and Peter Schartner. Application-level Security for ROS-based Applications. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4477–4482, 2016.

[16] Constanze Dietrich, Katharina Krombholz, Kevin Borgolte, and Tobias Fiebig. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1272–1289, New York, NY, USA, 2018. ACM.

[17] Roland Dóczi, Ferenc Kis, Balázs Sütő, Valéria Póser, Gernot Kronreif, Eszter Jósvai, and Miklos Kozlovszky. Increasing ROS 1.x Communication Security for Medical Surgery Robot. In *Proc. IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 4444–4449, 2016.

[18] eProsima. eProsima Fast RTPS, Mar. 2019. https://www.eprosima.com/index.php/products-all/eprosima-fast-rtps.

[19] Richard H. Epstein and Franklin Dexter. Influence of Supervision Ratios by Anesthesiologists on First-case Starts and Critical Portions of Anesthetics. *Anesthesiology: The Journal of the American Society of Anesthesiologists*, 116(3):683–691, 2012.

[20] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231, 1996.

[21] Sergio Garrido-Jurado, Rafael Muñoz Salinas, Francisco J. Madrid-Cuevas, and Rafael Medina-Carnicer. Generation of Fiducial Marker Dictionaries Using Mixed Integer Linear Programming. *Pattern Recognition*, 51(C):481–491, March 2016.

[22] Martin Giles. Triton is the World's Most Murderous Malware, and It's Spreading. *MIT Technology Review*, March 5, 2019. https://www.technologyreview.com/s/613054/cybersecurity-critical-infrastructure-triton-malware/.

[23] Megan Henney. Amazon Bear Repellent Accident this Week Wasn't Its First. *FOX Business*, December 7, 2018. https://www.foxbusiness.com/retail/amazon-bear-repellent-accident-this-week-wasnt-its-first.

[24] Brian Hernacki and William E. Sobel. Detecting Man-in-the-middle Attacks via Security Transitions, October 15, 2013. US Patent 8,561,181.

[25] Ling Huang, Anthony D. Joseph, Blaine Nelson, Benjamin I.P. Rubinstein, and J.D. Tygar. Adversarial Machine Learning. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, pages 43–58. ACM, 2011.

*[26]* Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access Pattern Disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *Proceedings of the Annual Network and Distributed System Security Symposium*, 2012. https://www.ndss-symposium.org/ndss2012/access-pattern-disclosure-searchable-encryption-ramification-attack-and-mitigation.

[27] Jongkil Kim, Jonathon M. Smereka, Calvin Cheung, Surya Nepal, and Marthie Grobler. Security and Performance Considerations in ROS 2: A Balancing Act. *CoRR*, abs/1809.09566, 2018. http://arxiv.org/abs/1809.09566.

[28] Dhilung Kirat, Jiyoung Jang, and Marc Ph. Stoecklin. DeepLocker: Concealing Targeted Attacks with AI Locksmithing. Black Hat USA, https://i.blackhat.com/us-18/Thu-August-9/us-18-Kirat-DeepLocker-Concealing-Targeted-Attacks-with-AI-Locksmithing.pdf, 2018.

[29] Laparoscopic.MD. Laparoscopic Trocars, Feb. 2019. https://www.laparoscopic.md/surgery/instruments/trocar.

[30] Xiao Li and Thenkurussi Kesavadas. Surgical Robot with Environment Reconstruction and Force Feedback. In *Proceedings of the 40th Annual International Conference of the IEEE Medicine and Biology Society*, pages 1861–1866, July 2018.

[31] Santiago Morante, Juan G. Victores, and Carlos Balaguer. Cryptobotics: Why Robots Need Cyber Safety. *Frontiers in Robotics and AI*, 2:23, 2015. https://www.frontiersin.org/article/10.3389/frobt.2015.00023.

[32] Seung Yeob Nam, Dongwon Kim, and Jeongeun Kim. Enhanced ARP: Preventing ARP Poisoning-based Man-in-the-middle Attacks. *IEEE Communications Letters*, 14(2):187–189, 2010.

[33] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference Attacks on Property-preserving Encrypted Databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 644–655. ACM, 2015.

[34] Object Management Group. DDS Portal - Data Distribution Services, 2018. https://www.omgwiki.org/dds/.

[35] Occipital. OpenNI 2 Downloads and Documentation, Mar. 2019. https://structure.io/openni.

[36] Open Robotics. ROS2 Overview, 2019. https://index.ros.org/doc/ros2/.

[37] Open Source Robotics Foundation. APIs - ROS Wiki, 2016. http://wiki.ros.org/APIs.

[38] Open Source Robotics Foundation. Robots That You Can Use with ROS, 2018. https://robots.ros.org/.

[39] Open Source Robotics Foundation. SROS, 2018. http://wiki.ros.org/SROS.

[40] Sean Palka and Damon McCoy. Fuzzing E-mail Filters with Generative Grammars and N-Gram Analysis. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., 2015. USENIX Association. https://www.usenix.org/conference/woot15/workshop-program/presentation/palka.

[41] Dan Petro and Ben Morris. Weaponizing Machine Learning: Humanity Was Overrated Anyway. DEF CON, https://www.defcon.org/html/defcon-25/dc-25-speakers.html#Petro, 2017.

[42] David Portugal, Samuel Pereira, and Micael Couceiro. The Role of Security in Human-robot Shared Environments: A Case Study in ROS-based Surveillance Robots. In *Proc. 26th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 981–986, Aug. 2017.

[43] David Portugal, Miguel Santos, Samuel Pereira, and Micael Coucerio. On the Security of Robotic Applications using ROS. In Roman V. Yampolskiy, editor, *Artificial Intelligence Safety and Security*, pages 279–289. Taylor & Francis, 2018.

[44] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: An Open-source Robot Operating System. In *Proceedings of the ICRA Workshop on Open Source Software*. IEEE, 2009.

[45] Erwin Quiring and Konrad Rieck. Adversarial Machine Learning against Digital Watermarking. In *Proceedings of the 26th European Signal Processing Conference (EUSIPCO)*, pages 519–523, Sep. 2018.

[46] Francisco J. Romero-Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. Speeded Up Detection of Squared Fiducial Markers. *Image and Vision Computing*, 76:38–47, 2018.

[47] Lea Schönherr, Katharina Kohls, Steffen Zeiler, Thorsten Holz, and Dorothea Kolossa. Adversarial Attacks against Automatic Speech Recognition Systems via Psychoacoustic Hiding. *CoRR*, abs/1808.05665, 2018. http://arxiv.org/abs/1808.05665.

[48] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Trans. Database Systems*, 42(3):19:1–19:21, July 2017.

[49] Senhance. The Senhance Surgical System, 2018. https://www.senhance.com/us/digital-laparoscopy.

[50] Aashish Sharma, Zbigniew Kalbarczyk, James Barlow, and Ravishankar Iyer. Analysis of Security Data from a Large Computing Organization. In *Proc. 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 506–517. IEEE, 2011.

[51] Aravind Sundaresan, Leonard Gerard, and Minyoung Kim. Secure ROS. Computer Science Laboratory, SRI International, http://secure-ros.csl.sri.com/, 2017.

[52] Intuitive Surgical. Da Vinci by Intuitive: Enabling Surgical Care to Get Patients back to What Matters, 2019. https://www.intuitive.com/en-us/products-and-services/da-vinci/.

[53] Thanh N. Tran, Klaudia Drab, and Michal Daszykowski. Revised DBSCAN Algorithm to Cluster Data with Dense Adjacent Clusters. *Chemometrics and Intelligent Laboratory Systems*, 120:92–96, 2013.

[54] Jeffrey Voas, Rick Kuhn, Constantinos Kolias, Angelos Stavrou, and Georgios Kambourakis. Cybertrust in the IoT Age. *Computer*, 51(7):12–15, July 2018.

[55] Ruffin White, Henrik I. Christensen, and Morgan Quigley. SROS: Securing ROS over the Wire, in the Graph, and Through the Kernel. *CoRR*, abs/1611.07060, 2016. http://arxiv.org/abs/1611.07060.