Analysis and Mitigation of Function Interaction Risks in Robot Apps

Yuan Xu
State Key Laboratory of Computer Architecture, Institute of Computing Technology, University of Chinese Academy of Sciences, Peng Cheng Laboratory

Tianwei Zhang
Nanyang Technological University

Yungang Bao
State Key Laboratory of Computer Architecture, Institute of Computing Technology, University of Chinese Academy of Sciences, Peng Cheng Laboratory

ABSTRACT

Robot apps are becoming more automated, complex and diverse. An app usually consists of many functions, interacting with each other and the environment. This allows robots to conduct various tasks. However, it also opens a new door for cyber attacks: adversaries can leverage these interactions to threaten the safety of robot operations. Unfortunately, this issue is rarely explored in past works.

We present the *first* systematic investigation about the function interactions in common robot apps. First, we disclose the potential risks and damages caused by malicious interactions. By investigating the relationships among different functions, we identify and categorize three types of interaction risks. Second, we propose RTron, a novel system to detect and mitigate these risks and protect the operations of robot apps. We introduce security policies for each type of risks, and design coordination nodes to enforce the policies and regulate the interactions. We conduct extensive experiments on 110 robot apps from the ROS platform and two complex apps (Baidu Apollo and Autoware) widely adopted in industry. Evaluation results indicated RTron can correctly identify and mitigate all potential risks with negligible performance cost. To validate the practicality of the risks and solutions, we implement and evaluate RTron on a physical UGV (Turtlebot) with real-word apps and environments.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; *Information flow control*; *Software security engineering*.

KEYWORDS

Robot apps; Function interaction; Risk analysis and mitigation

ACM Reference Format:
Yuan Xu, Tianwei Zhang, and Yungang Bao. 2021. Analysis and Mitigation of Function Interaction Risks in Robot Apps. In *24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '21), October 6–8, 2021, San Sebastian, Spain.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3471621.3471854

RAID '21, October 6–8, 2021, San Sebastian, Spain

ACM ISBN 978-1-4503-9058-3/21/10…$15.00
https://doi.org/10.1145/3471621.3471854

1 INTRODUCTION

The robotics technology is rapidly integrated into every aspect of our life. Different types of robots and applications were designed to assist humans with many dangerous or tedious jobs. A robot app usually consists of multiple processes (a.k.a. nodes), with each one focusing on one specific function, e.g., localization, path planning. They interact with each other to complete the end-to-end task.

To ease the development of robot apps, many companies expose interfaces of massive functions for their products (e.g. Ford Open XC [15], Dji Onboard SDK [8], UR Application Builder [5]). Developers can then use these functions to create new apps. Alternatively, public platforms are introduced, where functions are developed in a crowd-sourcing manner by third-party developers and distributed through the open-source function markets. The most mainstream platform is the Robot Operating System (ROS) [2], which provides thousands of open-source robot functions. Functions from this platform have been widely adopted in the research community and many commercial products, such as Dji Matrice 200 drone [8], PR2 humanoid [21] and ABB manipulator [19].

However, these functions can be the Achilles' Heel of robot apps, threatening the safety of robot operations. There are two reasons that facilitate this hazard. (1) Public platforms like ROS allow third-party developers to share their functions. Different from other well-developed app stores (e.g., mobile devices [3, 10], PCs [13, 31, 32], IoT [4, 11, 28]), the ROS platform does not enforce any security inspection over the submitted code. An adversary can easily upload malicious functions to the platform for users to download. (2) Function nodes in a robot app have dynamic and frequent interactions with each other and the physical environment. Even one malicious node can affect the states and operations of the entire app, leading to severe privacy breach and physical damages [48, 70]. For instance, Chrysler Corporation recalled 1.4 million vehicles in 2015 due to a software vulnerability in its Uconnect dashboard computers [1]. An adversary could exploit it to hack into a jeep remotely and take over the dashboard functions.

To ensure the safety of robot apps, it is critical to protect the interactions among various functions inside the apps. We are interested in two questions: *What potential risks and security incidents can a malicious interaction bring? How can we detect and mitigate malicious interactions?* Unfortunately, there are currently few studies focusing on the interactions in robot apps. Security analysis of interactions in IoT systems have been explored [39, 41, 42, 45, 53, 60, 61, 67, 83, 88]. As robot apps have more complex and distinct features, it is hard to apply the above methods to the robot ecosystem, as discussed in § 8.

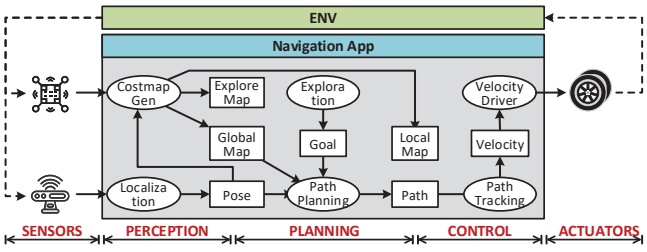


Figure 1: An example of the navigation app.

In this paper, we present the *first* study to explore the function interactions in common robot apps from the perspective of security and safety. We make three contributions to answer the above two questions. First, we analyze potential risks from those interactions in common robot apps. We classify these risks into three types. (1) *General Risk*: it happens when multiple function nodes share same states, and malicious nodes attempt to compromise the states by sending wrong messages. (2) *Robot-Specific Risk*: this is caused by the conflict between the robot's velocity and the frame rate of the image recognition function. (3) *Mission-Specific Risk*: this refers to the violation of users' expectation regarding the safe and secure behaviors of the robot system. We provide detailed analysis and examples to show the possible consequences of each risk.

Second, we introduce RTron, a novel system to detect and mitigate risks caused by suspicious interactions in robot apps. The core of RTron is a set of *coordination nodes*, which are used to regulate the interactions and enforce security policies. We design a coordinate node with some security policies to mitigate each type of risks. Specifically, RTron includes two stages. At the development stage, it generates the interaction graph from the source code of the robot app, and helps *developers* discover all high-risk function nodes, which may trigger potential malicious interactions. Based on the generated risk information, RTron deploys *coordination nodes* along with these high-risk function nodes. This is achieved without changing the original function node. At the operation stage, RTron deploys a security service to keep monitoring all the information from the coordination nodes. A visualized interface is provided to *end users* to observe the high-risk interactions. If a risk occurs, the corresponding coordination node will enforce the desired policy configured by users during the app launch to mitigate it.

Third, we conduct extensive experiments to evaluate the effectiveness, efficiency and practicality of RTron. (1) We select 110 robot apps from the ROS platform, covering 24 robots of 4 types. RTron can correctly identify all potential risks from three types of vulnerable interactions, with negligible overhead at both the offline and online stages. (2) We perform large-scale evaluations on more complex and practical robot apps: we select 2 apps from the ROS platform for the home and autorace scenarios, each containing 10 functions to perform 6 tasks; we also deploy 2 self-driving apps (Autoware [6] and Apollo [7]), which are widely adopted in the autonomous vehicle industry. RTron successfully identifies 198 high-risk interactions in these 4 apps, and mitigates them promptly and effectively. (3) We demonstrate a practical end-to-end attack with a physical robot (Turtlebot UGV) and environment, to demonstrate the feasibility and severity of malicious interactions in robot apps. We show this threat can be eliminated by RTron.

2 BACKGROUND & THREAT MODEL

2.1 Interaction in Robot Apps

Robot apps run on the embedded computer of a robot device to interpret sensory data collected from the environment, and make the corresponding action decisions. The workflow of a robot app can be represented as an *interaction graph*, where each node represents a certain function, and edges represent the dependencies of the functions in this app. Figure 1 shows a navigation app as an example. This robot app is composed of three major processing stages [76]: (1) *Perception*: the robot extracts estimated states of the environment and the device from raw sensory data. It uses the `Localization` node to determine the device position, and the `CostmapGen` node to model the surroundings. (2) *Planning*: the robot determines the long-range actions. It uses the `Path Planning` node to find the shortest path, and the `Exploration` node to search for all accessible regions. The `Exploration` node also exposes an external service for users to launch a navigation mission. (3) *Control*: the robot processes the execution action and forwards these motions to the actuators. It uses the `Path Tracking` node to produce velocity commands following the planned path, and the `Velocity Driver` node to convert the velocity to instructions for the motor to drive the wheels.

One big feature of robot apps is the high interactions among various function nodes in the workflow. Based on the triggered events, the interactions can be classified into two groups:

Direct interaction (solid line). This denotes the interaction between two functions (ellipses), which are directly connected in the workflow and sharing common robot states (squares). Robot states are defined as the collection of all aspects and knowledge of the device that can impact future behaviors [78], e.g., position, orientation, explored maps. The computation of one function can change some robot states, which will affect the computation of another function. For instance, in Figure 1, the action of `Path Planning` is triggered by the event that `Localization` generates the robot's current position and orientation. Then the two nodes have direct interaction over the robot states of position and orientation.

Indirect interaction (dotted line). This refers to the dependency of two functions, which are not connected in the workflow, but can interact with each other via the environmental context. One node in the app can issue actions to change the environmental context (e.g., obstacles, space, etc.), which will further influence another node. In the navigation app, the functions in the *Control* stage generate commands to control the robot to change the physical environment. This triggers the functions in the *Perception* to conduct new computations. For instance, the map created by the `CostmapGen` function depends on the action from the `Path Tracking` function. As a result, these two function nodes are indirectly interacted, although they are not directly connected in the workflow.

Note that Figure 1 is just an abstract interaction graph. An actual robot app can have a very complex interaction graph with large numbers of nodes and interactions. Figure 15 in Appendix A gives an interaction graph for a real-world home-based robot app.

2.2 Robot App Platform

In robotics, the most popular app platform is Robot Operating System (ROS) [2]. Both the research community and industry widely adopt ROS as the foundation or the testbed for their apps, such as Dji

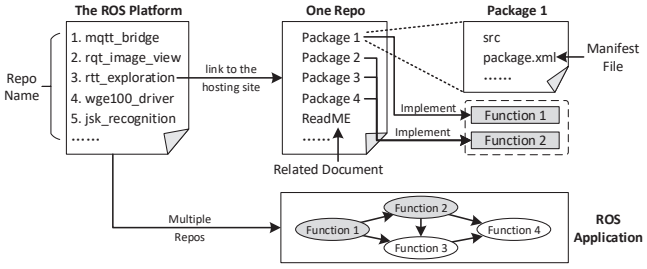


Figure 2: The relationship among the app, repo, package and function in ROS

Matrice 200 drone [8], PR2 humanoid [21] and ABB manipulator [19]. In this paper, we mainly focus on the ROS platform. Our methods and conclusions can be generalized to other platforms as well.

The ROS platform offers two kinds of services. First, it provides *robot core libraries*, which act as the middleware between robot apps and hardware. These core libraries support hardware abstraction, message passing mechanisms and device drivers for hundreds of sensors and motors. Second, the ROS platform maintains thousands of *robot code repositories* (a.k.a. repos) for distributed version control, code management and sharing. As shown in Figure 2, the platform stores a list of ROS indexes (i.e. repo names), and each index is linked to the source code of this repo in the hosting site (e.g. GitHub, BitBUcket, GitLab). A repo commonly consists of one or multiple ROS packages. The developers can add their repos to the ROS platform through sending a *pull* request to the ROS maintainer. If it succeeds, both the repos and included packages can get specific indexes for other developers to download and use.

A robotic function can be implemented by one or multiple packages. It means one repo can have two or more functions. These functions are then integrated with functions from other repos or customized by users to form a ROS application. This work shows that untrusted repos from the ROS platform can significantly threaten the robot apps built from them.

Development and operation of robot apps. Figure 3 illustrates the key concepts and components in the lifecycle of robot app development and operation. First, the design of the app is decomposed into several necessary functions. Among them core functions (white ellipses) need to be customized by the developer, while non-core functions (black ellipses) can be downloaded from ROS code repos (❶). Then the developer uses ROS core libraries to organize these functions as an app workflow (❷) and deploys the app to the robot (❸). Each function is abstracted as a ROS node and connected with others through ROS *Topics*. The ROS topics are many-to-many named buses that store the robot or environment states. Each topic is implemented by the *publish-subscribe* messaging protocol: some nodes can subscribe to a topic to obtain relevant data, while some nodes can publish data to a topic.

The robot communicates with end users through ROS *Services*. The ROS services are a set of interfaces of the robot app exposed to end users. Each service is implemented by the *Remote Procedure Call* (RPC) protocol and allows users to launch tasks or adjust function parameters. Once the robot receives a mission from the user's phone (❹), it executes the mission and interacts with the surrounding environment at runtime (❺). The user will receive the notification from the robot when all tasks are completed (❻).

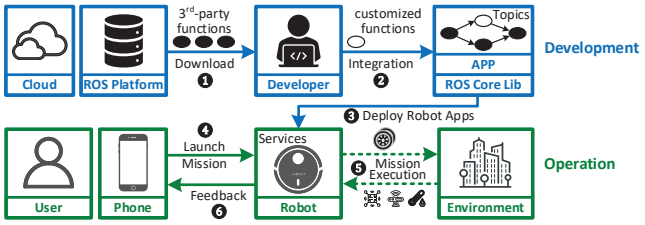


Figure 3: The lifecycle of robot app development (blue parts) and operation (green parts).

2.3 Threat Model and Problem Scope

In this paper, we consider a threat model where some nodes of a robot app are untrusted. Those adversarial nodes aim to compromise the robot's operations, forcing it to perform dangerous actions. This can result in severe security and safety issues to machines, humans and environments [43, 49, 50].

This threat model is drawn from four observations. First, the ROS platform is open for everyone to upload and share their code repos. Different from app stores of other ecosystems [3, 4, 10, 11, 13, 28, 31, 32], the ROS platform *does not have any security check over the submitted code*. As a result, an adversarial developer can insert malicious code to a repo and publish it to the ROS platform for other users to download. This has been highlighted in the design document of ROS2 Robotic Systems Threat Model [18]: "*third-party components releasing process create additional security threats (third-party component may be compromised during their distribution)*". We also confirm the feasibility and practicality of this threat with an end-to-end attack demonstrated in § 7. Second, the quality of third-party function code is not guaranteed. A lot of functions in the ROS platform are in a lack of coding standards or specifications. They may also contain software bugs that can be exploited by an adversary to compromise the nodes at runtime [50, 65]. By inspecting the latest commit logs in the Robot Vulnerability Database [16], 17 robot vulnerabilities and 834 bugs (e.g., no authentication, uninitialized variables, buffer overflow) were discovered in the repos of 51 robot components, 37 robots and 34 vendors in the ROS platform. Most of them are still not addressed yet. Third, the high interactions among nodes in a robot app can amplify the attack damage. If an adversary controls one node, it is possible that he can affect other nodes directly or indirectly, and then the entire app. The existence of untrusted nodes can cause data races or deadlocks when the synchronization is not well handled. Finally, this threat model is widely adopted in prior works regarding ROS security [38, 51, 56, 85].

Given this threat model, our goal is to design a methodology and system, which can identify and mitigate the safety risks caused by the malicious nodes inside robot apps. For instance, an adversary can flood the path planning node to block other nodes publishing goals or increase the speed so that the robot would be too fast to miss the target searching objects or obstacles in the surroundings. We focus on the protection of node interactions (both direct and indirect) instead of the operation of individual nodes. We further assume the underlying OS and ROS core libraries are trusted: the

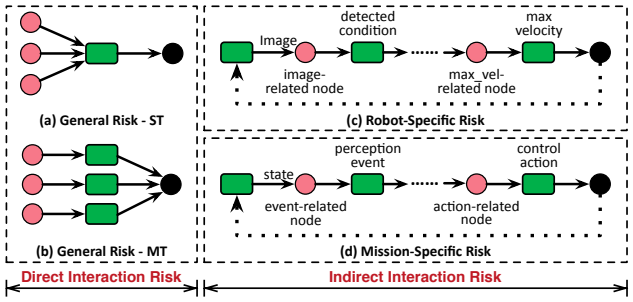


Figure 4: Three types of interaction risk.

operational flow and data transmission are well protected, and the isolation scheme is correctly implemented so the malicious nodes are not able to hijack the honest ones or the privileged systems. How to enhance the security of the ROS core libraries [50, 52, 54, 80] and mitigate vulnerabilities from networks [36, 44, 64, 71], sensors [40, 47, 47, 68, 72, 74, 75, 77, 79, 81, 82, 84, 87], actuators [46, 55] and controllers [69] are orthogonal to our work.

3 RISK ANALYSIS

We analyze safety risks caused by malicious function nodes and interactions. We classify these risks into three categories (Figure 4 and Table 2). We describe how each risk can incur unexpected behaviors to threaten the robot safety.

3.1 General Risk (GR)

GR is caused by a direct interaction. It occurs when multiple function nodes share the same robot states. If one node is malicious, it can intentionally change the robot states to wrong values to affect the robot operation. Based on the interaction graph, there are two conditions to trigger the GR. First, two or more function nodes are connected to the same successor node, and at least one of them is untrusted. Second, the transmitted message types among the above function nodes need to be the same. This guarantees that all these nodes share the same robot state through the direct interaction.

According to the number of topics, GR can be further divided into two types. (1) General Risk with Single Topic (GR-ST): multiple high-risk nodes publish to one same topic, subscribed by the successor node (Figure 4a). (2) General Risk with Multiple Topics (GR-MT): both the indegree and outdegree of the topic are equal to 1. There can be multiple parallel topics with the same message type subscribed by the successor function (Figure 4b).

3.2 Robot-Specific Risk (RSR)

RSR happens in an indirect interaction, due to the conflict behaviors related to the robotic mobility characteristic. This mobility feature requires the robot to recognize real-time environment conditions (e.g. obstacle avoidance, traffic light) and react to them promptly. The robot's maximal velocity is determined by its reaction time, which further depends on two factors [37, 62]. The first factor is the processing time for collision avoidance, which is the end-to-end latency from obstacle detection to velocity control. The second factor is the frame rate of the `Image Recognition` function. The faster the robot is, the larger frame rate this function requires to respond to the rapid changes of the environment. In this paper, we only focus on the second factor as the processing latency is the safety issue of the internal function node (i.e. `Path Tracking`) rather than the interaction between two nodes.

Figure 4c shows the mechanism of RSR. There are two types of high-risk function nodes: (1) the image-related node is used to understand the current detected conditions through image recognition. (2) The max_vel-related node outputs the maximal velocity value to the corresponding topic based on the current condition. These two nodes affect each other via an indirect interaction (dotted line). The maximal velocity and image frame rate should satisfy certain conditions to guarantee the robot can function correctly. If either node is malicious and produces anomalous output (too large maximal velocity or too small frame rate), the requirement can be compromised, bringing catastrophic effects in some tasks.

3.3 Mission-Specific Risk (MSR)

MSR refers to the violation of users' expectations regarding the safe and secure behaviors of a robot system. It exists in the indirect interaction between an event-related node and action-related node (Figure 4d), when there are conflicts between them, regulated by some scenario-specific rules. Although some GRs and RSRs may also lead to the violation of these rules, the causes and mitigation strategies are totally different. So it is necessary to discuss MSR separately. There are two types of high-risk nodes in MSR: (1) the event-related ones include all the nodes in the *Perception* domain except `Preprocessing`. The robot uses those nodes to understand the conditions of the physical environment. (2) The action-related ones include all the nodes in the *Control* domain which can directly interact with the actuator drivers. They are used to actively change the actual states of both the robot and environment. If either of these nodes are malicious, the robot and task can be compromised with unexpected consequences.

The rules to prevent MSR are determined by the missions and usage scenarios, which are usually specified by users. Table 1 lists some examples of MSRs and the corresponding rules in four scenarios. (1) In a domestic context, robots are designed to manage various human-centric tasks, e.g., house cleaning, baby-sitting. They are required not to disturb human's normal life. (2) In a warehouse context, industrial robots are introduced to achieve high automation and improve the productivity, such as manipulator and autonomous ground vehicle (AGV). These robots are required to complete each subtask correctly, efficiently and safely. (3) In a city context, autonomous vehicles and delivery robots move at high speeds in the transportation system, and handle complex events from outdoor dynamic environment. Thus, they need to obey the transportation rules and ensure the safety of passengers and public assets. (4) Robots are also deployed in many specialized scenarios to conduct professional missions. For example, rescue robots are used to search for survivors or extinguish fires. Medical robots are used in hospitals to diagnose and treat patients. Military robots are designed in battlefields to destroy enemies or constructions. These robots need to follow the rules related to their specific missions.

Table 1: Examples of Mission-Specific Risks and Rules.

Scenario	Description
Domestic	The companion robot must send an alert when a user is in danger.
	The robotic vacuum must be turned off when a user is sleeping.
Warehouse	The manipulator must not grasp objects that exceed its limited weight.
	The AGV must recharge when the battery level is below a threshold.
City	The mobile vehicle must follow the traffic rule.
	The mobile vehicle must maintain a safe distance with passengers.
Specialized	The firefighter robot must send an alert when detecting the wounded.
	The precision of the surgery robot must be above a specified threshold.

3.4 Summary of Risks from Each Domain

An arbitrary malicious node in the robot app can incur the above risks. We discuss the potential risks and consequences caused by malicious functions in each domain.

Perception. If a node in the *Perception* domain is untrusted, the robot states will be estimated as wrong values. Following the direct interactions, the robot will take anomalous actions, which violate the rules of MSR. Moreover, since the `Recognition` function typically adopts sensor fusion to reduce uncertainty caused by the physical limit of different sensors, such threat can cause GR as well.

For instance, an autonomous vehicle is navigating in a highway. A malicious `Preprocessing` function intentionally sends wrong sensory data to the `Object Recognition` function to cause optical illusions, e.g., recognizing a turn right sign as a stop sign. This will violate the traffic rule: "vehicles cannot stop in a highway".

Planning. A malicious node in the *Planning* domain can interrupt the current task, or reset the robot states to wrong values. In a common robot app, there can be multiple `Global Planner` functions for different goals based on various events from the `Recognition` functions. This gives the malicious node chances to win the competition against other goals and compromise the robot states (GR). Besides, the malicious node can also directly modify the goal to make the robot take anomalous actions in a specific event (MSR).

For instance, a robot vacuum is executing the cleaning task in a living room. The `Global Planner` function is compromised and controlled by an adversary to set a new destination goal as the master bedroom for stealing privacy. This can violate a possible MSR rule: "the robot vacuum cannot enter the bedroom". If the robot does not have enough power to clean the master bedroom, this will violate the MSR rule: "the AGV must recharge when the battery level is below a specified threshold." (Table 1).

Control. If a function in the *Control* domain is malicious, the adversary can launch attacks in three ways. First, the function can interrupt or suspend other actions from different interactions (GR). Second, it can increase the velocity to cause failures of image-related recognition functions through the indirect interaction (RSR). Third, it can directly control the robot to take unexpected actions in a specific scenario (MSR).

For instance, in a task of searching dangerous goods or wounded persons, the robot device receives images through the equipped camera at a certain frame rate. If the max_vel node is malicious and intentionally increases the maximal velocity, there will be no or less correlation between adjacent frames. The `Image Recognition` function may fail to process each frame promptly, and frames containing safety-related information (e.g. drug, thief) can be missed.

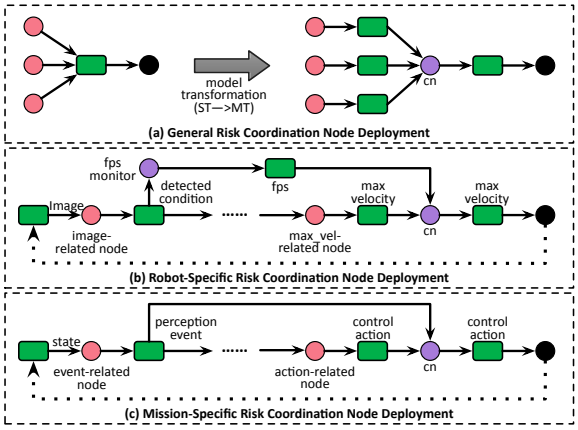


Figure 5: Three types of coordination nodes (purple circles).

4 MITIGATION METHODOLOGY

We present a novel methodology to mitigate the malicious function interactions. The core of our solution is a set of *coordination nodes* (§ 4.1) and *security policies* (§ 4.2), as summarized in Table 2.

4.1 Coordination Node

The coordination nodes are deployed inside the robot apps to regulate the interactions and enforce the desired security policies. They are designed to be general for different types of robots, function nodes and risks. Developers can deploy them into apps without modifying the internal function code. Users can adjust configurations based on their demands. We design three types of coordination nodes, to mitigate three types of risks respectively (Figure 5).

General Risk Coordination Node (GRCN). This node is inserted between the high-risk nodes and their successor node (Figure 5a). The published topics of each high-risk node need to be remapped to the subscribed topic of this GRCN to create new data flows, and the published topic of the GRCN need to be mapped to the subscribed topic of the successor node. Thus, the GRCN can control each data flow from the high-risk nodes based on various policies.

Robot-Specific Risk Coordination Node (RSRCN). This node needs to coordinate the conflict between the image-related node and max_vel-related node (Figure 5b). We use the same method to insert the RSRCN between the max_vel-related node and its successor node. To collect the frame rate from the image-related node, we insert a fps_monitor node to subscribe to the detected condition topic published by the image-related node. This fps_monitor node measures the frequency of the triggered event and publishes the frame rate to the fps topic. The RSRCN subscribes to this fps topic and uses it as reference for max velocity adjustment.

Mission-Specific Risk Coordination Node (MSRCN). This node needs to allow/block the actions taken under wrong conditions (Figure 5c). Thus, it is deployed between each action-related node and its successor, and subscribes to all perception event topics of event-related nodes. In this way, the MSRCN can collect all perception events in the app and obtain the control of each action. It is worth noting that there can be multiple GRCNs for each interaction, but the numbers of both RSRCN and MSRCN are always one.

Table 2: Summary of risks, threats and mitigation for function interactions.

Risk	Domain Threat	Coordination Node	Executor	Policy	Parameter	Description
GR	Perception, Planning, Control	GRCN	Developer	Block	Block Bit	Allow/block the action of chosen flow.
				FIFO_Queue	Timeout	Choose the action based on fifo order with time limit.
				Priority_Queue	Timeout, Priority	Choose the action based on priority order with time limit.
				Preemption	Priority	Choose the action based on priority order.
RSR	Control	RSRCN	Developer	Block	Block Bit	Allow/block the velocity control action of chosen flow.
				Safe	Threshold, Priority	Adjust max velocity based on fps data.
			End User	Constrain	Max_vel_limit	Limit adjustable max velocity limit with a user-defined value.
MSR	Perception, Planning, Control	MSRCN	End User	Block	Block Bit	Allow/block the action of chosen flow.

4.2 Security Policies

To mitigate the malicious interactions in an app, each type of coordination nodes implements a set of policies. Table 2 lists the policies we have built along with the descriptions and parameters for GRCN, RSRCN and MSRCN. Each policy needs to be configured by either the developer or end user, as shown in the "Executor" column.

GRCN Policies. GRCN aims to coordinate data flows from different high-risk nodes. We use four types of policies to adapt to different scenarios. Specifically, the block policy is used when the user wants to stop the current action immediately in case of emergency. When multiple high-risk nodes publish control commands, the preemption policy will choose the action with the highest priority. For example, both the `Safe Control` and `Path Tracking` nodes publish velocity to the `Mobile Driver` node. However, the safe control action should be taken first because it is responsible for ensuring user's safety. FIFO_Queue and Priority_Queue policies are used for high-risk nodes with high requirements of completion time, such as search, rescue and obstacle avoidance.

RSRCN Policies. RSRCN aims to resolve the conflicts between data flows from the image-related (*iflow*) and max_vel-relate (*vflow*) nodes. We use three types of policies to adjust the maximal velocity of the robot. Block policy allows/blocks the action from *vflow* and does not affect the action from *iflow*. Safe policy uses thresholds to bridge the maximal velocity with fps. Based on the fact that a higher velocity requires a faster processing capability, we assume the maximal velocity is proportional to the fps. Then the threshold serves as a scale factor and can be configured by users. Constrain policy sets a maximal velocity limit to ensure safety in complex and dynamic environments. This is particularly useful when users want the robots to work at low speeds psychologically even though they drive within safe speed ranges.

MSRCN Policies. MSRCN aims to coordinate the conflicts between the data flows from the event-related node (*eflow*) and action-relate node (*aflow*). We only adopt block policy to decide whether the action should be taken under some specific conditions. However, the block bits of *eflow* and *aflow* are different. Bit 0/1 in *aflow* denotes that the actions are allowed/blocked, while Bit 0/1 in *eflow* represents whether the condition event is triggered or not. Thus, end users can control all the actions under arbitrary conditions.

To reduce the complexity of configuring our methodology for unexperienced end users, we delegate part of the policy selection and parameter configuration tasks to the developers. It is reasonable because some risks are derived from the race condition while the others are caused by falling short of user's expectation. Specifically, the developers enforce appropriate policies for each GRCN and set the corresponding parameters. Moreover, the developers also preset the parameters in the block and safe policies for RSRCN based on the robot's characteristics. On the other hand, the end users only have the control of policy selection in RSRCN and MSRCN. The parameters they need to configure are just max_vel_limit in RSRCN and block bit in MSRCN. Table 2 shows the role of end users and developers for each policy (the "Executor" column).

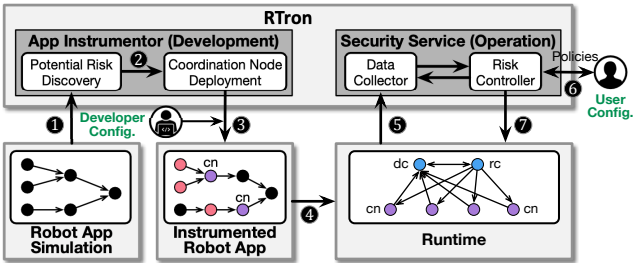


Figure 6: RTron system overview.

5 SYSTEM DESIGN

We design RTron, a novel end-to-end system equipped with the above mitigation. Given a potential vulnerable robot app, the developer first utilizes RTron to add necessary coordination nodes to the app without modifying the original function node, and set up some security policies. Then the end user can safely launch the patched app on the robot, and configure other policies before the task starts. Figure 6 gives the overview of RTron. It consists of two components: (1) an *App Instrumentor* for developers to detect potential risks in robot apps and deploys coordination nodes (§ 5.1); (2) a *Security Service* that visualizes and configures the coordination nodes to mitigate risks at runtime (§ 5.2).

5.1 App Instrumentor

The goal of this module is to instrument the target app's source code to make it compatible with RTron. It patches an app with certain coordination nodes to collect events and actions from high-risk function nodes, and guard the robot at runtime. Two subcomponents are introduced to identify high-risk function nodes, and the locations to deploy the coordination nodes, respectively.

Potential Risk Discovery. This submodule is designed to help developers identify high-risk function nodes in a robot app. It first simulates the lifecycle of the target app and automatically generates the interaction graph offline. Then it traverses all function nodes (black circles in Figure 6) in the graph and identifies three types of high-risk function nodes: GR node RN_{gr}, RSR node RN_{rsr} and MSR node RN_{msr} (❶). Algorithm 1 describes our identification strategy. We conclude one rule to discover each type of risky nodes:

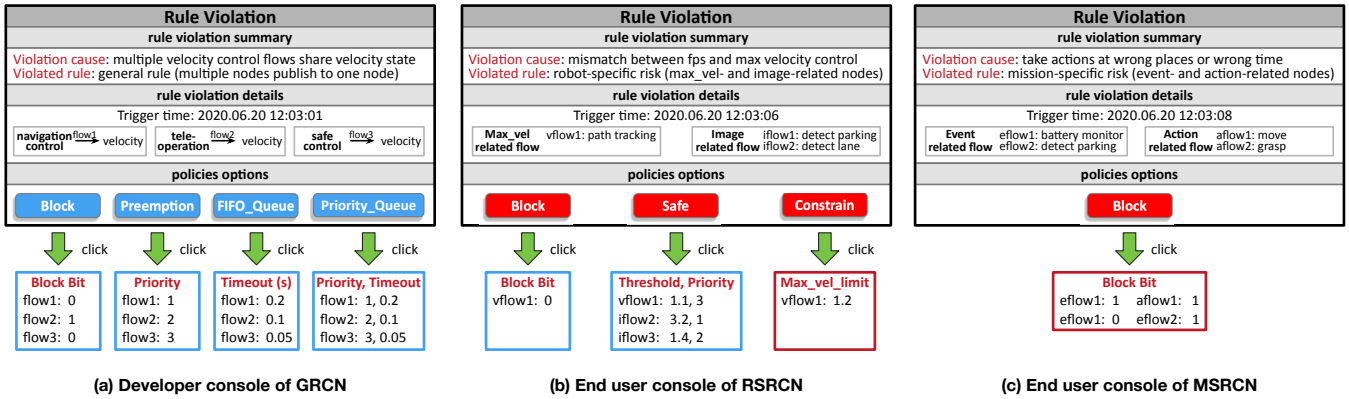


Figure 7: Developer and end user console of each risk in RTron. The red solid rectangle denotes a button for the end users. The blue/red box represents policy-related configuration parameters for the developers/end users.

Algorithm 1: Algorithm for Potential Risk Discovery

Input: N ▷ A set of nodes in a robot app
T ▷ A set of topics in a robot app
N_j^p ▷ A set of nodes publish to the topic j
T_i^s ▷ A set of topics subscribed by the node i
T_i^p ▷ A set of topics published by the node i
Output: RN ▷ Risk nodes in a robot app

1 **foreach** *topics* $t_j \in T$ **do**
2 **if** $num(N_j^p) > 1$ **then**
3 $RN_{gr}^{ST} \leftarrow \{N_j^p\}$;
4 **if** $('max\_vel' \in t_j.name) \wedge (t_j.type == 'std\_msgs/Float64')$ **then**
5 $RN_{rsr}^{max} \leftarrow \{N_j^p\}$;
6 **foreach** *string* $s_n \in EVENT\_MSG\_TYPE$ **do**
7 **if** $(s_n \in t_j.type) \vee ('detect' \in t_j.name)$ **then**
8 $RN_{msr}^{event} \leftarrow \{N_j^p\}$;
9 **foreach** *string* $s_n \in ACTION\_MSG\_TYPE$ **do**
10 **if** $s_n \in t_j.type \vee ('goal' \in t_j.name)$ **then**
11 $RN_{msr}^{action} \leftarrow \{N_j^p\}$;
12 **foreach** *node* $n_i \in N$ **do**
13 sort node's subscriptions T_i^s by $T_i^s.type$;
14 **foreach** *subscription* $s_k \in T_i^s$ **do**
15 **if** $s_k.type == s_{k+1}.type$ **then**
16 $RN_{gr}^{MT} \leftarrow \{n_i\}$;
17 **foreach** *subscription* $s_k \in T_i^s$ **do**
18 **if** $s_k.type == 'sensor\_msgs/Image'$ **then**
19 **foreach** *publication* $p_m \in T_i^p$ **do**
20 **foreach** *string* $s_n \in RECOG\_TOPIC\_NAME$ **do**
21 **if** $s_n \in p_m.name$ **then**
22 $RN_{rsr}^{image} \leftarrow \{n_i\}$;

GR Rule: we identify the topics in the graph whose indegree is greater than 1. All nodes that publish to these identified topics are denoted as RN_{gr}^{st} with single topic (Lines 1-3). The node with more than one subscribed topics of the same message type can be integrated to RN_{gr}^{mt} with multiple topics (Lines 12-16).

RSR Rule: to identify the image-related node RN_{rsr}^{image} and max_vel-related node RN_{rsr}^{max}, RTron checks the topic name and type of each subscribed or published message (Lines 4-5,17-22). It searches the key words (e.g., 'detect', 'people' and 'face') in the RECOG_TOPIC_NAME string list. Evaluations in § 6 indicate this key word searching can effectively identify the RSR nodes.

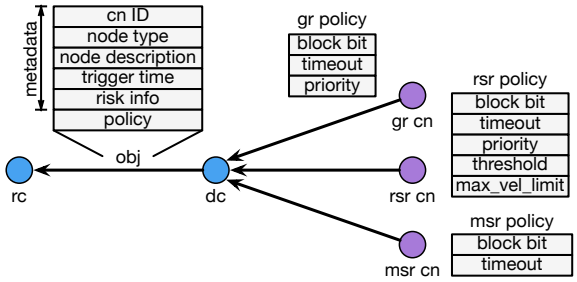


Figure 8: Risk model of three types of risks in *Data Collector*.

MSR Rule: to identify the event-related node RN_{msr}^{event} and action-related node RN_{msr}^{action}, RTron checks if the message type of each topic (Lines 6-11) is in the EVENT_MSG_TYPE or ACTION_MSG_TYPE lists since message types typically use standard ROS naming conventions [20]. The complete lists of EVENT_MSG_TYPE and ACTION_MSG_TYPE are presented in Tables 3 and 4.

Coordination Node Deployment. The collected information of potential risks is used to configure the coordination node setting(❷). This includes a set of topics and parameters. Topics represent the state transition between two function nodes: the subscribed and published topics specify the predecessor and successor nodes of each coordination, respectively. The parameters are used to expose an interface to the end user for configuring each policy. With these configuration files, a *Coordination Node Deployment* submodule is designed to deploy coordination nodes into the app automatically (❸). Meanwhile, the developers check the details of the risks, select the optional policies for GRCN and configure related parameters.

Figure 7(a) shows an example of GRCN. GRCN monitors velocity data from three risky nodes: `Navigation Control`, `Tele-operation` and `Safe Control`. The data transmission of each node is marked as flow1, flow2 and flow3. The developer can select the Priority_Queue policy after the app is launched, and set flow3 from `Safe Control` as the highest priority, indicating its velocity action should be always taken first. However, if the coordination

Table 3: Description of EVENT_MSG_TYPE.

Message Type	Description
sensor_msgs/ BatteryState	Measurement of the battery state (voltage, charge, etc).
sensor_msgs/ Temperature	Measurement of the temperature.
sensor_msgs/ RelativeHumidity	Defines the ratio of partial pressure of water vapor to the saturated vapor pressure at a temperature.
sensor_msgs/ MagneticField	Measurement of the Magnetic Field vector at a specific location.
sensor_msgs/ FluidPressure	Measurement of the pressure inside of a fluid (air, water, etc), atmospheric or barometric pressure.
sensor_msgs/ NavSatFix	Measurement for any Global Navigation Satellite System (latitude, longitude, etc).
sensor_msgs/ Illuminance	Measurement of the single photometric illuminance.
nav_msgs/ Odometry	Measurement of an estimate of a position and velocity in free space (pose, twist, etc).

Table 4: Description of ACTION_MSG_TYPE.

Actuator	Message Type	Description
Mobile	geometry_msg/ Twist	This expresses the velocity in free space broken into its linear and angular parts.
Manipulator	control_msgs/ FollowJoint TrajectoryAction	This defines the joint trajectory to follow.
Speaker	audio_common _msg/AudioData	This defines the audio data to speak.

node cannot receive the responding actions before the user-defined timeout (i.e. 0.2s), it will transmit the velocity action of flow2 with the second priority.

5.2 Security Service

This module aims at visualizing and mitigating risks of malicious interactions at runtime. It consists of two subcomponents deployed along with the robot app.

Data Collector. When the robot executes the app within the environment, all coordination nodes in the instrumented app keep forwarding their information to this submodule (❹). Such information is stored as a risk model, which consists of metadata and a set of policy parameters (❺). As shown in Figure 8, the metadata records basic information of a coordination node, including its ID, node type, node description, trigger time and risk information. They manage each coordination node and visualize to the users for risk display and policy configuration.

Risk Controller. This submodule visualizes risk information and enforces policies from users to each coordination node. Right after the app is launched on the robot, the *Risk Controller* obtains all the information of each coordination node from the *Data Collector*. It then configures each coordination node by sending user-defined policy parameters (❼). When the *Data Collector* receives an event and the corresponding coordination nodes' actions at runtime, the *Risk Controller* evaluates them against a collection of security policies. Some policies are mandatory, while some are optional, depending on the real-world demands (e.g. task or scenario) of end users.

The *Risk Controller* provides an interface for end users to check the details of risks and select the optional policies (❽). Figure 7 presents the user consoles for three types of coordination nodes. There are three components in each console. (1) The *rule violation summary* component shows the violation cause and rule of this risk. (2) The *rule violation details* component presents the trigger time and detailed information, e.g., potential malicious nodes, flows. (3) The *policies options* component provides optional policy to either developers or end users in the different stages of RTron. Note that the end users only have full control of policy selection for RSRCN and MSRCN, and parameter configurations for two specific policies.

Taking RSRCN as an example (Figure 7(b)). End users can check the current violation information and reset the corresponding policy parameters at runtime. When a robot moves from an obstacle-free environment (e.g., Highway) to a complex environment (e.g. downtown area), users can select the Constrain policy in an RSRCN to limit the robot's maximal velocity.

5.3 Policy Configuration

To sum up, the protection is enforced by both the developer and end user with the following steps:

Risk Identification. In the development stage, the developer first launches the target robot app in the simulator, and uses just an one-line command "rosrisk-search [gr|rsr|msr|all]" to automatically identify potential risks in the app. Based on the identified information, the developer needs to configure the name of predecessor nodes and successor nodes in each coordination node configuration file. Note that there is no need to modify the source code of the original app in this step. Each coordination node would be launched and deployed into the app automatically.

Risk Mitigation. Risks are mitigated in both the development stage and operation stage. As shown in Figures 7(a) and (b), the developer can choose the GRCN policy (blue button), and customize GRCN and part of RSRCN parameters for each policy (blue square). In the operation stage, the end users can get the console of RSRCN and MSRCN. They can choose RSRCN and MSRCN policy (red button), and customize MSRCN and part of RSRCN parameters for each policy (blue square).

6 EVALUATION

We aim to answer the following questions:

- Can RTron effectively detect three types of interaction risks? What is the relationship between the interaction risks and task characteristics in each robot app? (§ 6.1)
- How many coordination nodes are required to deploy in a typical robot app? How to configure the policy for an end user under various environmental contexts? (§ 6.2)
- What is the performance overhead of RTron? (§ 6.3)

Testbed. We study 110 open-source apps from the ROS showcase website [17], covering 24 different robots including mobile base (MB), mobile manipulator (MM), micro aerial vehicle (MAV) and humanoid robot (HR). Table 5 summarizes the categories of these

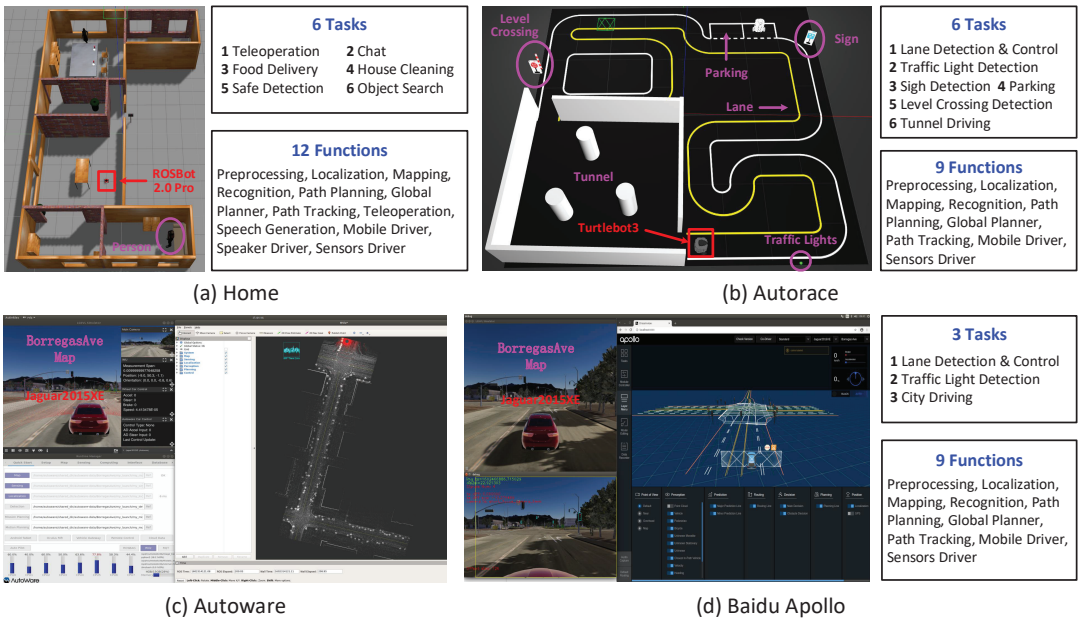


Figure 9: Four simulated scenarios in the Gazebo/LGSVL.

apps, numbers and the applicable robot types. In addition, we also perform analysis of more complex apps (Figure 9):

- *Home scenario*: home-based apps and robots are used to accompany people and conduct housework. These tasks include teleoperation, chat, food/drink delivery, cleaning, safe detection, and object search. We use four ROS apps (Remote Control, Face/Person Detection, Object Search and Voice Interaction) of RosBot 2.0 Pro [22] to develop one home app (Figure 9a).
- *AutoRace scenario [30]*: this type of apps is designed for competition of autonomous driving robot platforms. To ensure that the robot can drive on the track safely, there are six necessary missions for the robot to execute, including lane detection & control, traffic light detection, sign detection, parking, level crossing detection and tunnel driving. We use the open-source Autonomous Driving app of Turtlebot3 [29] which can realize all six tasks in the autorace scenario (Figure 9b).
- *Autonomous driving scenario*: we consider two mainstream self-driving apps: Autoware [6] and Apollo [7], which have been fully deployed and tested in physical autonomous vehicles. These two apps are more complex than the AutoRace scenario, with a richer set of self-driving modules composed of sensing, computing, and actuation capabilities (Figure 9c and 9d).

Experimental Setup. Since this paper focuses on the software risks in robot apps, we mainly use simulation to validate our solution. Implementation and evaluation on physical robots will be demonstrated in § 7. We choose the Gazebo simulator [9] and ROS Kinetic in the home and autorace scenarios, which run on a server equipped with 1.6GHz 4-core Intel i5 processor and Nvidia MX110 GPU. In the autonomous driving scenario, we use the LGSVL simulator [12] with ROS Indigo for Apollo 3.5, and ROS Melodic for Autoware 1.14, running on a server with 4.2GHz 8-core Intel i7 and Nvidia GTX 1080 GPU. We use Rviz [27] to visualize 3D information from both the simulator and robot apps.

Table 5: Analysis of open-source robot apps from the ROS showcase website [17].

App Categories	# of apps	Robot Type	Example Robot
Remote Control	23 (20.8%)	MB, MM, HR, MAV	Caster
Panorama	2 (1.8%)	MB	Turtlebot3
2D/3D Mapping	8 (7.3%)	MB	Xbot
Navigation	22 (20%)	MB, MM, MAV	Tiago++
SLAM	11 (10%)	MB	Roch
Exploration	5 (4.5%)	MB	Turtlebot2
Follower	8 (7.3%)	MB	Magni Silver
Manipulation	8 (7.3%)	MM	LoCoBot
Face/Person Detection	8 (7.3%)	MB, MM, MAV	ARI
Object/Scene Detection	5 (4.5%)	MM	Tiago
Object Search	1 (1%)	MM	ROSbot 2.0 PRO
Gesture Recognition	3 (2.7%)	HR, MAV	COEX Clover
Voice Interaction	5 (4.5%)	MB, HR	Qtrobot
Autonomous Driving	1 (1%)	MB	Turtlebot3

6.1 Risk Identification

Single-functional Apps. We successfully extract all the GRs and MSRs from all 110 open-source apps. GRs are identified by checking the nodes and topics based on their topology relationship. Some GRs are ignored when they publish messages to the log/visualization topic, which will not bring risks to the robot app. MSRs are identified by inspecting if the standardized topic types are matched.

Different from the GR rule, the RSR rule involves the identification of specific topic names and types. We choose 15 image-related apps (e.g., Face/Person Detection, Object/Scene Detection, Object Search, Autonomous Driving) and 1 max_vel-related app (Autonomous Driving). We successfully discover all 20 image-related and 4 max_vel-related RSRs from these apps.

Table 6: Examples of high-risk nodes in the Home and AutoRace apps.

Scenario	Risk Type	High-Risk Nodes	Sub Topic Name	Sub Topic Type	Pub Topic Name	Pub Topic Type	Pub Node
Home	GR-ST	/move_base	-	-	/cmd_vel	geometry_msgs/Twist	/gazebo
		/teleop_twist_keyboard	-	-	/cmd_vel	geometry_msgs/Twist	/gazebo
	GR-MT	/gazebo	-	-	/camera/depth/image_raw	sensor_msgs/Image	/find_object_3d
		/gazebo	-	-	/camera/rgb/image_raw	sensor_msgs/Image	/find_object_3d
	RSR-Image	/find object 3d	/camera/rgb/image raw	sensor_msgs/Image	/objects	std msgs/Float32MultiArray	/search manager
	MSR-Event	/move_base	/odom	nav_msgs/Odometry	-	-	-
	MSR-Action	/rosbot_tts	-	-	audio_common_msgs/AudioData	/rosbot_audio/audio	/rosbot_audio
AutoRace	GR-ST	/detect_tunnel	-	-	/move_base_simple/goal	geometry_msgs/PoseStamped	/move_base_simple/goal
		/rviz	-	-	/move_base_simple/goal	geometry_msgs/PoseStamped	/move_base_simple/goal
	GR-MT	/detect/lane	-	-	/detect/lane	std_msgs/Float64	/control/lane
		/detect_traffic_light	-	-	/control/max_vel	std_msgs/Float64	/control/lane
	RSR-Image	/detect_sign	/camera/image_compensated	sensor_msgs/Image	/detect/traffic_sign	std_msgs/UInt8	/core_mode_decider
	RSR-Max_vel	/detect_parking	-	-	/control/max_vel	std_msgs/Float64	/control_lane
	MSR-Event	/core_node_controller	/detect/tunnel_stamped	std_msgs/UInt8	-	-	-
	MSR-Action	/detect_tunnel	-	-	/cmd_vel	geometry_msgs/Twist	/gazebo

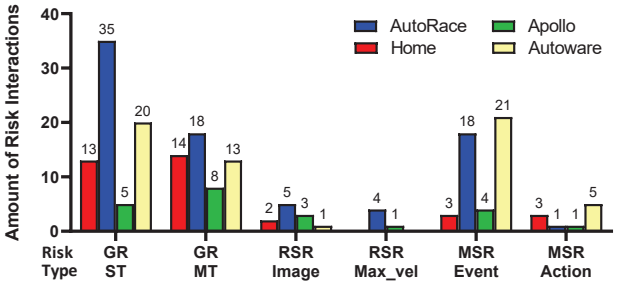


Figure 10: Numbers of high-risk nodes in four robot apps.

Multi-functional Apps. RTron is also scalable for analysis of more complex apps. RTron successfully identifies 198 risk interactions in the four target apps. Figure 10 lists the numbers of extracted nodes with respect to each risk type. We can observe the numbers of risk interactions in the autorace (blue bar) and autoware (yellow bar) apps are larger than home (red bar) and apollo (green bar) apps, although the home app has the largest number of functions. This is caused by the differences in the internal structure of each robot app. In the home scenario, each task is relatively independent. However, in the autorace and autoware apps, all tasks are organized as a monolithic component to control the robot to drive safely. To achieve this, these two apps need to recognize various scenes from sensory images and take the corresponding actions. Consequently, the high dependency among those tasks increases the number of GRs. Moreover, the requirement of image and scene recognition increases the number of image-related RSRs and event-related MSRs. Apollo is a special case where the number of topics is far smaller than the other apps, thus the number of risks is also the smallest (Table 9). Table 6 gives examples of the identified high-risk node for each type in Home-based and AutoRace app. Texts marked in red are for risk identification in our system.

6.2 Risk Mitigation

CN Analysis. RTron uses the extracted risk information to deploy CNs. For GRs, the number of GRCNs depends on the number of high-risk interactions linked to the same node. Thus, RTron checks the GR information of "Pub Node" and deploys the GRCN between high-risk nodes and their pub node. For RSRs, since RSRCNs directly publish velocity messages to the `Mobile Driver` function, the number of RSRCN is always 1. The subscriptions of RSRCN is related to the number of image-related nodes and max_vel-related nodes.

Table 7: Numbers of CNs in four complex robot apps.

Scenario	GRCN			RSR		MSR CN
	Perception	Planning	Control	FMN	CN	
Home	8	3	1	2	1	1
AutoRace	16	2	4	5	1	1
Apollo	4	1	1	3	1	1
Autoware	11	3	2	1	1	1

Table 8: High-risk interacted topics and features of three GRCN types in the home app.

CN Type	Interacted Topics	Feature
Perception	'/explore_server/status', '/move_base/status', 'tf', 'tf_static', '/camera/rgb/image_raw', '/camera/depth/image_raw', '/move_base/global_costmap/footprint', '/move_base/local_costmap/footprint'	State Parallelization
Planning	'/move_base/goal', '/move_base/cancel', '/move_base_simple/goal'	Goal Queuing
Control	'/cmd_vel'	Action Preemption

Besides, as described in § 4.1, each image-related node should be assigned to an fps_monitor node to generate the processing rate of the image recognition process. So the number of required fps_monitor nodes depends on the number of image-related nodes. For MSRs, the number of MSRCNs is 1, as all event-related and action-related nodes publish corresponding messages to the MSRCN, which then sends the action message to all related actuator driver nodes.

Table 7 lists the numbers of three types of CNs in the four robot apps. GRCNs account for a large portion of the total added nodes. Due to a large number of RSR image-related interactions, the autorace app has more fps_monitor nodes than the home app.

Policy Selection. RTron implements a variety of policies for three types of CNs. How to select the appropriate policy for each CN is critical for the secure operation of robot apps. We use the home app as an example to illustrate the guideline for policy selection.

GRCN: this is designed to coordinate direct high-risk interactions between multiple connected nodes. Based on the types of interacted topics, we classify GRCN into three categories: perception, planning and control. As shown in Table 8, the messages of interacted topics in perception are related to the sensory information (e.g. images) or

Table 9: Processing time of potential risk discovery.

Application	Node Number	Topic Number	Processing Time (s)		
			GR	RSR	MSR
Teleoperation [26]	4	17	0.114	0.113	0.057
Voice Interaction [33]	6	7	0.035	0.035	0.011
Mapping [25]	6	25	0.308	0.299	0.152
Navigation [24]	8	63	0.764	0.727	0.498
Exploration [23]	10	84	1.12	1.086	0.753
Home	21	125	3.121	3.199	1.927
AutoRace [30]	25	112	4.075	4.049	2.105
Apollo [7]	21	39	0.631	0.606	0.306
Autoware [6]	38	218	2.945	2.931	1.747

preprocessed robot states (e.g. footprints, status). Typically, multiple messages with the same type are published to the same target node, and processed in parallel for either sensor fusion or state monitoring. Thus, there is no contention among these messages.

Messages of the interacted topics in planning or control contend with each other to get the long-term and instant control of the robot. Specifically, when a message of a new planning goal is received, the robot must first complete the previous goal before executing the current one. For example, an object search task is launched after the `search_manager` node publishes a goal to the `/move_base_simple/goal` topic. An adversary can use a malicious `rviz` node to send another arbitrary destination to this topic. The object search task will be immediately interrupted and then the robot is controlled to reach the designated position. Thus, a GRCN with the 'FIFO_Queue' or 'Priority_Queue' policy can delay such malicious actions without task interruption.

Different from the planning messages, the control messages need to control the robot immediately. End users can select the 'Preemption' policy of GRCN for coordination. For instance, the malicious `teleop_twist_keyboard` node can flood the `/cmd_vel` topic while the robot is following a planned path to the destination. Then the topic receives the messages from both `teleop_twist_keyboard` and `move_base` nodes simultaneously, which causes the robot to switch velocity in the two target directions. By assigning the highest priority to the move_base-related velocity control interaction (i.e. `/cmd_vel`), the `move_base` node can control the robot first.

RSRCN: end users are not recommended to set the 'Block' or 'Safe' policy. These two options should be chosen by app developers after extensive evaluations. Instead, users can choose the 'Constrain' policy to set a maximal velocity value to limit the robot's speed. This is very effective and safe, especially when the robot's working environment is highly complex and dynamic, and the task completion time is not very critical. For example, if an adversary compromises the `move_base` node and increases the robot's speed to a dangerous level, this can cause a potential traffic accident. By setting an appropriate threshold in the 'Safe' policy or max_vel_limit in the 'Constrain' policy, the robot will slow down its speed without object detection failures.

MSRCN: although there is only one policy option, users can customize different rules to allow/block the actions of specific robots under specific conditions. Taking the home app as an example, the MSRCN receives messages from three event-related topics (`/objects`, `/person_detector/detections`, `/odom`) and two action-related topics (`/audio/audio`, `/cmd_vel`). Users can set a rule to disallow the robot's movement when it detects the target object. This can identify and mitigate the interruption of the object search task caused by the malicious `rviz` node mentioned above.

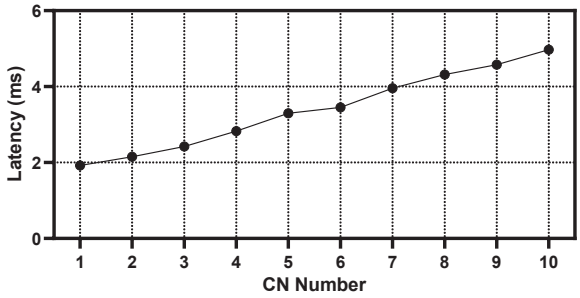


Figure 11: Overhead of CNs in an end-to-end data flow.

6.3 Performance Overhead

Offline overhead. We evaluate the risk discovery stage of RTron in terms of processing time for identifying high-risk nodes in a robot app. Table 9 reports the performance results of 9 robot apps with different numbers of topics and nodes. We repeat each experiment for 20 times to calculate the average latency. We conclude that the risk discovery has negligible overhead as an offline process. The results also show that the processing time is affected by the number of topics and nodes. This is because the risk identification depends on the traversal of either nodes or topics (Algorithm 1). Specifically, there are two iterations in the process of both GR and RSR discovery and one iteration of topics in the process of MSR discovery. Thus, discovering GR takes similar time as RSR, which is longer than MSR. One exception is the autorace app, which has the largest processing time, but fewer nodes and topics than the home app. This is because there are more high-risk GR interactions in the autorace app (Table 7), which add extra work (i.e. related topic type and name match) in the node iteration process.

Runtime overhead. This includes the overhead from the coordination nodes and security service. The security service is only responsible for risk monitoring and policy configuration of each coordination node, without any interference on the execution of the robot app. Much like IoT policy enforcement systems [42, 53], we ignore the overhead of this process since users manually configure the policy for each CN only at the mission launch stage or scenario change condition. The coordination nodes are distributed among function nodes in the robot app, which can increase the end-to-end latency from the perception to the control stages. Although there are dozens of nodes in a typical robot app, these nodes work in a parallel multi-flow mode. To achieve real time, typically each data flow includes fewer than 10 nodes. So we consider the overhead of end-to-end latency within 10 coordination nodes. As shown in Figure 11, the extra latency incurred by 10 coordination nodes is around 5ms. This is trivial even for the autonomous driving app with the strongest real-time constraint: according to the industry standards published by Mobileye [73] and design specifications from Udacity [14], the latency for processing tragic condition in an autonomous driving app should be within 100 ms, which is far larger than the overhead of coordination nodes.

7 CASE STUDIES IN THE REAL WORLD

To demonstrate the practicality of the considered threats and proposed solution, we implement and evaluate several scenarios in a physical device, i.e., Turtlebot3. Figure 12 shows our settings and

real-world environment. The Turtlebot3 is an open-source mobile base equipped with a Raspberry Pi CPU@1.3GHz, 1 GB memory and a 360 Laser Distance Sensor (LDS), running ubuntu 16.04 and ROS kinetic. It is connected to a server (Intel i7 CPU@4.2GHz with 16GB of RAM) for computation offloading and mission launching.

7.1 Attack Method

We develop two normal tools `/tb3_safe_control` and `tb3_monitor` to monitor and control the robot's movement. We insert some malicious codes in these tools which send wrong control commands. The `/tb3_safe_control` provides commands for safe teleoperation with different input devices. It use LaserScan information to estimated the distance between the robot and obstacles, and stop the robot's movement within a customized safe distance. The `tb3_monitor` package provides commands to monitor nodes' information and robot's states in real time. We encapsulate these tools into two ROS packages and successfully upload them to the ROS platform as a developer. *This validates our threat model that an adversary can easily share malicious packages in the ROS platform.* Next, we download these two packages as another developer, and implement them on the Turtlebot device. Below we describe the malicious behaviors and how our system can mitigate them with three cases. To avoid raising ethical concerns, we add an extra trigger such that the attacks happen only when the MAC address of the robot matches a predefined one. This ensures that the malicious package will not affect normal users.

Figure 13 illustrates the attack and its consequences. The malicious code of GR attack is added in the `/tb3_safe_control` package. The malicious codes of RSR and MSR attack are all hidden in the `tb3_monitor` package. It's worth noting that we add specific triggering logics (Lines 2) in each attack to avoid raising ethical concerns. The triggering condition is the success match between the default MAC address and local host MAC address. Since the MAC address is unique of different devices, the malicious codes can only work in our robotic devices. Moreover, we set time matching process to make the attack launch at specific time, other than at the beginning. This can make attacks more hidden.

Figure 13(a) shows the related code snippets in the GR attack.The gr_attack function is invoked by a callback function of LaserScan Topic. In each iteration, the function starts by searching the gr-related vulnerable node, i.e. '`move_base`'. If exists, it means the move-related control topic '`cmd_vel`' exists and the robot is executing a navigation task with a great probability. Thus, we send a Twist-type move command with -0.2 z-axis angular velocity to the victim topic. This would lead to the robot suddenly turn right and crash to the obstacles while navigating in a collision-free path.

Figure 13(b) and (c) present the malicious code snippets in the RSR and MSR attack respectively. Both rsr_attack and msr_attack functions are invoked while each traversal of all topics of one node. Specifically, once the '`control/max_vel`' topic exists, the malicious process would send a max velocity control command with 2 m/s to the victim topic. Similarly, if the '`move_base_simple/goal`' topic exists, a goal with a malicious location will be launched to the victim topic and the robot would move to the dangerous destination.

7.2 Evaluation Results

GR Case. The `/tb3_safe_control` node generates malicious velocity commands during the robot's navigation at certain moments. In Figure 12a, the robot plans a straight route in the corridor. During its movement, the `/move_base` node computes the real-time velocity and publishes it to the `/cmd_vel` topic to drive the robot to the destination. Due to the shared state between these two nodes, the malicious node compromises the robot and creates a crash through publishing continuous "turn right" commands to the shared topic. In RTron, the developer can choose *Preemption* policy in the GRCN and set different priorities to each flow at development stage. Since this operation is offline, the overhead can be ignored. Then the malicious node is not able to interrupt the normal navigation behaviors in this case.

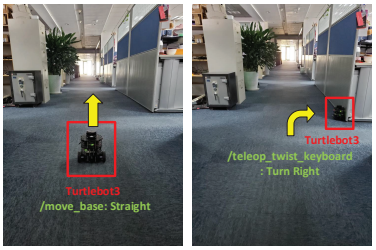
MSR Case. The `tb3_monitor` node sends malicious goal commands during the robot's navigation at certain moments. As shown in Figure 12b, it generates a wrong destination in an unstable area far away from the wireless access point. As part of the computations is offloaded to the remote server, the robot running into this area will lose network connection, and malfunction. In our experiment, after the destination is changed, the robot navigates into the unstable area and finally stops under the poor network condition. In RTron, the developer can use position from the `/odom` topic to implement a function node to check whether the robot moves in the unstable area. This node is connected with MSRCN and marked as an eflow. In this way, any suspicious destination within the unstable area would be blocked. End user can choose *Block* policy in the MSRCN and set different parameters to each flow at runtime. Since this one-time action is taken when launching the robot app, the potential overhead does not matter.

RSR case. Considering the potential physical damages caused by vehicle's high speeds, the RSR case is implemented in the simulator. The `tb3_monitor` node sends malicious max velocity configuring commands druing the robot's navigation at specific moments. It increases the max velocity value through publishing the malicious messages to the `/control_max_vel` topic. In Figure 14, the initial max velocity is 0.22m/s and the robot moves safely. At one moment, this value is increased to 2m/s. Then the robot moves too fast to detect the obstacle and a collision occurs. In RTron, we choose *Constrain* policy in the RSRCN and set 0.22 to the vflow. In this way, the max velocity of the robot is fixed at 0.22m/s and cannot be changed by the attacker.

8 RELATED WORKS

Robotic Security. Existing research on robotic security has mainly focused on traditional security issues in robot systems, e.g., network communication [52, 54, 80], denial-of-service attacks [50] and software vulnerabilities [57–59, 63, 69]. In addition, adversaries can also spoof the sensory data ([40, 47, 68, 72, 74, 75, 77, 79, 81, 82, 84, 87]), fake the actuator signals [46], or tamper with the micro-controller input [69].

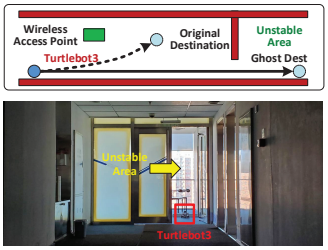
In this paper, we focus on a new type of security issue in robot apps, caused by malicious interactions. We are the first to demonstrate the feasibility and severity of this threat, as well as a possible defense solution against it.



GR Nodes	Pub Topic Name
/move_base	/cmd_vel
/tb3_safe_control	/cmd_vel

GRCN Policy	Preemption
GRCN Flows	Parameter
flow1	2
flow2	1

(a) GR Case



MSR Nodes	Pub Topic Name
[Event] /turtlebot3_core	/odom
[Action] /tb3_monitor	/move_base_simple/goal

MSRCN Policy	Block
MSRCN Flows	Parameter
eflow	1
aflow	0

(b) MSR Case

Figure 12: GR and MSR experiments on turtlebot3.

```
def gr_attack():
  if mac == HOST_MAC:
    if cur_time_min == 15 and
       node_exist('move_base'):

      twist = Twist()
      twist.linear.x = twist.linear.y
          = twist.linear.z = 0.0
      twist.angular.x = twist.angular
          .y = 0.0
      twist.angular.z = -0.2

      gr_atk_pub = rospy.Publish(
          'cmd_vel', Twist,
          queue_size=10)
      gr_atk_pub.publish(twist)
```

(a) Malicious codes of GR attack

```
def rsr_attack():
  if mac == HOST_MAC:
    if cur_time_min == 30 and
       topic_exist('control/max_vel')
      :

      max_vel = Float64()
      max_vel.data = 2

      rsr_atk_pub = rospy.Publisher(
          'control/max_vel', Float64,
          queue_size=1)
      rsr_atk_pub.publish(max_vel)
```

(b) Malicious codes of RSR attack

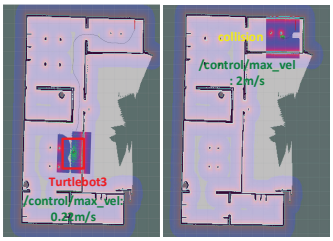
```
def msr_attack():
  if mac == HOST_MAC:
    if cur_time_min == 45 and
       topic_exist('move_base_simple/
    goal'):

      goal = PoseStamped()
      goal.header.stamp = now()
      goal.header.frame_id = "map"
      goal.pose.position = MAL_LOC

      msr_atk_pub = rospy.Publisher(
          'move_base_simple/goal',
          PoseStamped, queue_size=1)
      msr_atk_pub.publish(goal)
```

(c) Malicious codes of MSR attack

Figure 13: Malicious codes in our `tb3_safe_teleop` and `tb3_monitor` packages.



RSR Nodes	Pub Topic Name
[Max_vel] /tb3_monitor	/control_max_vel
[Image] /move_base	/detect/objects

RSRCN Policy	Constrain
RSRCN Flows	Parameter
vflow	0.22

Figure 14: RSR experiment on Turtlebot3 in the simulation.

Interaction Risk Mitigation. Prior works studied the interaction risks in IoT apps [39, 41, 42, 45, 53, 60, 61, 67, 83, 88]. Users adopt operation rules following the "If-This-Then-That" (IFTTT) trigger-action programming paradigm [66, 86] to express automation behaviors among IoT devices. These methods translate the rules to the interaction graph, and verify if conflicts or policy violations can occur between interactions.

There are three major differences between the interaction risks of robot apps and IoT apps. (1) For interaction modeling, robot apps not only inherit all the interactions from IoT apps, but also enjoy robot-specific ones, e.g., direct interactions via sharing internal states, indirect interactions caused by mobility. Robot apps need to cooperate with multiple functions, and require more complicated rules than the IFTTT model in IoT apps. (2) For risk identification, IoT apps are implemented by verifying if the interaction between different rules violates user-defined policies. However, robot apps have not only such risks (MSR), but also new ones (GR and RSR) due to data competition and mobility. (3) For risk mitigation, different from the simple "allow/block" policy adopted in IoT works, coordination of each type of risks needs a set of different configurable policies to mitigate malicious function interactions. All these distinct features of robot apps require new studies about the risk analysis and mitigation solutions, as we present in this paper.

9 DISCUSSION AND FUTURE WORK

Graph-based Analysis Scheme. We build graphs to analyze the interaction risks in robot apps. This graph-based analysis technique has been used in other appified platforms (e.g., smartphone, IoT and SDN) to identify potential malicious interaction as well. However, the potential risks in those platforms are different from robot apps. In smartphones, attacks occur when the users make uninformed decisions during app installation, or grant wrong resource requests without considering the contextual information at runtime [34]. In SDN, one unprivileged app can trick another privileged app through the shared control plane [35]. This is similar to GR in robot apps that one node manipulates the shared states (i.e. topic) to attack the other. In IoT platforms, the malicious interaction leverages *physical* channels to indirectly launch attacks [53]. However, different from RSR and MSR, the IoT devices do not have the mobility feature, and the usage scenario is usually restricted (e.g., smart home). In summary, the malicious interactions in robot apps are more complex and diverse, adding new challenges for risk analysis.

Policy Design. The security policies in robot apps are different from those in other domains (e.g., smartphone, IoT and SDN) from

two perspectives. First, we adopt distributed coordination nodes for policy enforcement, rather than centralized permission-based systems in other domains. This is because the centralized master-based robot system has been replaced by the distributed P2P system in the ROS2 due to its single-point-of-failure and real-time constraints. Second, the policy responses in other platforms are mainly block, warn or none. This is also applied to the MSR in robot apps. However, they cannot handle the malicious flows in GR and RSR. New actions (e.g., coordinating execution order of each flow, adjusting the data of specific flows) are needed.

As we discuss in § 6.2, there are some regularities to select policies based on the risk type and domain type of the coordination node. Thus, while processing a new robot applications, these rules can help developers to design correct polices for GRCN and parameters for GRCN and RSRCN. For other polices and parameters, developers can use our identified information to provide end users a detailed policy selection instruction. Since the number of RSRCN and MSRCN is limited, this is not a hard work.

Limitations. RTron also suffers from some limitations. First, we assume the structure of the target app follows the standard interaction graph and categorization. If the developer customizes the app based on totally different designs, then it may fail to discover potential risks. Second, RTron requires the source code of the robot app for risk identification and mitigation. It becomes challenging when the source code is not available. We will explore the risk analysis and protection of closed-source robot apps as future work.

10 CONCLUSION

Function interaction provides great flexibility and convenience for robot app development. However, it also introduces potential risks that can threaten the safety of robot operations. This is exacerbated by the fact that current robot app stores do not provide security inspection over the function packages. We present the first study towards the safety issues caused by suspicious function interaction in robot apps. We introduce a novel end-to-end system and method to enforce security policies and protect the function interactions in robot apps. We hope this study can open a new direction for robotics security, and increase people's awareness about the importance of function interaction protection.

ACKNOWLEDGMENTS

We thank our Shepherd Dr. Hamed Okhravi and anonymous reviewers for their valuable comments. This work was supported in part by Key-Area Research and Development Program of Guangdong Province (NO.2020B010-164003), the National Natural Science Foundation of China (Grant No. 62090020), Youth Innovation Promotion Association of Chinese Academy of Sciences (2013073), and the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No. XDC05030200), Singapore MoE AcRF Tier 1 RG108/19 (S) and NTU-Desay Research Program 2018-0980.

REFERENCES

[1] 2015. After Jeep Hack, Chrysler Recalls 1.4M Vehicles for BugFix. https://www.wired.com/2015/07/jeep-hack-chrysler-recalls-1-4m-vehicles-bug-fix/.
[2] 2019. Open source robot operating system. http://www.ros.org/.
[3] 2020. App Store. https://www.apple.com/ios/app-store/.
[4] 2020. Apple HomeKit. https://developer.apple.com/homekit/.
[5] 2020. Application Builder. https://www.universal-robots.com/builder/.
[6] 2020. The Autoware.AI Project. https://github.com/Autoware-AI/autoware.ai.
[7] 2020. Baidu Apollo. https://github.com/ApolloAuto/apollo.
[8] 2020. Dji Onboard SDK. https://developer.dji.com/onboard-sdk/.
[9] 2020. Gazebo 3D Robot Simulator. http://gazebosim.org/.
[10] 2020. Google Play. https://play.google.com/store/.
[11] 2020. Google Weave Project. https://developers.google.com/weave/.
[12] 2020. LGSVL Simulator. https://www.lgsvlsimulator.com/.
[13] 2020. The Mac App Store. https://www.apple.com/uk/osx/apps/app-store//.
[14] 2020. An Open Source Self-Driving Car. https://www.udacity.com/self-driving-car/.
[15] 2020. OpenXC Platform. http://openxcplatform.com/.
[16] 2020. Robot Vulnerability Database (RVD). https://github.com/aliasrobotics/RVD/.
[17] 2020. Robots that you can use with ROS. https://robots.ros.org/.
[18] 2020. ROS 2 Robotic Systems Threat Model. https://design.ros2.org/articles/ros2_threat_model.html.
[19] 2020. ROS ABB Package. http://wiki.ros.org/abb/.
[20] 2020. ROS Messages. http://wiki.ros.org/Messages/.
[21] 2020. ROS PR2 Package. http://wiki.ros.org/Robots/PR2/.
[22] 2020. ROSbot 2.0 PRO. https://store.husarion.com/collections/dev-kits/products/rosbot-pro/.
[23] 2020. RosBot Exploration App. https://husarion.com/tutorials/ros-tutorials/8-unknown-environment-exploration/.
[24] 2020. RosBot Navigation App. https://husarion.com/tutorials/ros-tutorials/7-path-planning/.
[25] 2020. RosBot SLAM App. https://husarion.com/tutorials/ros-tutorials/6-slam-navigation/.
[26] 2020. RosBot Teleoperation App. https://husarion.com/tutorials/ros-tutorials/3-simple-kinematics-for-mobile-robot/.
[27] 2020. Rviz 3D visualization tool for ROS. https://www.stereolabs.com/docs/ros/rviz/.
[28] 2020. Samsung SmartThings. https://www.smartthings.com/.
[29] 2020. Turtlebot3. https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.
[30] 2020. Turtlebot3 AutoRace. https://emanual.robotis.com/docs/en/platform/turtlebot3/autonomous_driving.
[31] 2020. Ubuntu Appstore. https://ubuntu.com/blog/tag/appstore/.
[32] 2020. Windows Apps - Microsoft Store. https://www.microsoft.com/en-us/store/apps/windows/.
[33] 2020. Xiaoqiang Voice Interaction App. https://community.bwbot.org/topic/492/.
[34] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick D. McDaniel, and Matthew Smith. 2016. SoK: Lessons Learned from Android Security Research for Appified Software Platforms. In *IEEE Symposium on Security and Privacy (S&P)*.
[35] Samuel Jero Benjamin E. Ujcich, Anne Edmundson, Qi Wang, Richard Skowyra, James Landry, Adam Bates, William H. Sanders, Cristina Nita-Rotaru, and Hamed Okhravi. 2018. Cross-App Poisoning in Software-Defined Networking. In *ACM Conference on Computer and Communications Security (CCS)*.
[36] Tamara Bonaci, Jeffrey Herron, Tariq Yusuf, Junjie Yan, Tadayoshi Kohno, and Howard Jay Chizeck. 2015. To Make a Robot Secure: An Experimental Analysis of Cyber Security Threats Against Teleoperated Surgical Robots. In *CoRR abs/1504.04339*.
[37] Behzad Boroujerdian, Hasan Genc, Srivatsan Krishnan, Wenzhi Cui, Aleksandra Faust, and Vijay Janapa Reddi. 2018. MAVBench: Micro Aerial Vehicle Benchmarking. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[38] Benjamin Breiling, Bernhard Dieber, and Peter Schartner. 2017. Secure communication for the robot operating system. In *IEEE Systems Conference (SysCon)*.
[39] Lei Bu, Wen Xiong, Chieh-Jan Mike Liang, Shi Han, Dongmei Zhang, Shan Lin, and Xuandong Li. 2018. Systematically Ensuring the Confidence of Real-Time Home Automation IoT Systems. *ACM Transactions on Cyber-Physical Systems (TCPS)* 2, 3 (2018), 22:1–22:23.
[40] Yulong Cao, Chaowei Xiao, Benjamin Cyr, Yimeng Zhou, Won Park, Sara Rampazzi, Qi Alfred Chen, Kevin Fu, and Z. Morley Mao. 2019. Adversarial Sensor Attack on LiDAR-based Perception in Autonomous Driving. In *ACM Conference on Computer and Communications Security (CCS)*.
[41] Z. Berkay Celik, Patrick D. McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. In *Annual Technical Conference (ATC)*.
[42] Z. Berkay Celik, Gang Tan, and Patrick D. McDaniel. 2019. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *Annual Network and Distributed System Security Symposium (NDSS)*.
[43] Cesar Cerrudo and Lucas Apa. 2017. Hacking Robots Before Skynet. *IOActive Website* (2017).
[44] Jiyang Chen, Zhiwei Feng, Jen-Yang Wen, Bo Liu, and Lui Sha. 2019. A Container-based DoS Attack-Resilient Control Framework for Real-Time UAV Systems. In *Design, Automation, and Test in Europe (DATE)*.
[45] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. 2018. Cross-App Interference Threats in Smart Homes: Categorization, Detection and Handling. In

CoRR abs/1808.02125.

[46] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Xinyan. 2018. Detecting Attacks Against Robotic Vehicles: A Control Invariant Approach. In *ACM Conference on Computer and Communications Security (CCS).*

[47] Drew Davidson, Hao Wu, Robert Jellinek, Vikas Singh, and Thomas Ristenpart. 2016. Controlling UAVs with Sensor Input Spoofing Attacks. In *Workshop on Offensive Technologies (WOOT).*

[48] Nicholas DeMarinis, Stefanie Tellex, Vasileios P. Kemerlis, George Dimitri Konidaris, and Rodrigo Fonseca. 2019. Scanning the Internet for ROS: A View of Security in Robotics Research. In *International Conference on Robotics and Automation (ICRA).*

[49] Tamara Denning, Cynthia Matuszek, Karl Koscher, Joshua R. Smith, and Tadayoshi Kohno. 2009. A spotlight on security and privacy risks with future household robots: attacks and lessons. In *Ubiquitous Computing (UbiComp).*

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

[51] Bernhard Dieber, Severin Kacianka, Stefan Rass, and Peter Schartner. 2016. Application-level security for ROS-based applications. In *International Conference on Intelligent RObots and Systems (IROS).*

[52] Bernhard Dieber, Severin Kacianka, Stefan Rass, and Peter Schartner. 2016. Application-level security for ROS-based applications. In *International Conference on Intelligent RObots and Systems (IROS).*

[53] Wenbo Ding and Hongxin Hu. 2018. On the Safety of IoT Device Physical Interaction Control. In *ACM Conference on Computer and Communications Security (CCS).*

[54] Roland Dóczi, Balázs Süto Ferenc Kis, Valeria Poser, Gernot Kronreif, Eszter Josvai, and Miklos Kozlovszky. 2016. Increasing ROS 1.x communication security for medical surgery robot. In *IEEE International Conference on Systems, Man and Cybernetics (SMC).*

[55] Fan Fei, Zhan Tu, Ruikun Yu, Taegyu Kim, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. 2018. Cross-Layer Retrofitting of UAVs Against Cyber-Physical Attacks. In *IEEE International Conference on Robotics and Automation (ICRA).*

[56] David Ke Hong, John Kloosterman, Yuqi Jin, Yulong Cao, Qi Alfred Chen, Scott A. Mahlke, and Z. Morley Mao. 2020. AVGuardian: Detecting and Mitigating Publish-Subscribe Overprivilege for Autonomous Vehicle Systems. In *European Symposium on Security and Privacy (EuroS&P).*

[57] Michael Hooper, Yifan Tian, Runxuan Zhou, Bin Cao, Adrian P. Lauf, Lanier Watkins, William H. Robinson, and Wlajimir Alexis. 2016. A review on cybersecurity vulnerabilities for unmanned aerial vehicles. In *Military Communications Conference (MILCOM).*

[58] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. 2019. Rvfuzzer: Finding input validation bugs in robotic vehicles through control-guided testing. In *USENIX Security Symposium (USENIX Security 19).*

[59] C. G. Leela Krishna and Robin R. Murphy. 2017. A review on cybersecurity vulnerabilities for unmanned aerial vehicles. In *IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR).*

[60] Chieh-Jan Mike Liang, Börje F. Karlsson, Nicholas D. Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. 2015. SIFT: building an internet of safe things. In *International Symposium on Information Processing in Sensor Networks (IPSN).*

[61] Chieh-Jan Mike Liang, Zhao Li Lei Bu, Junbei Zhang, Shi Han, Börje F. Karlsson, Dongmei Zhang, and Feng Zhao. 2015. Systematically Debugging IoT Control System Correctness for Building Automation. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments (BuildSys@SenSys).*

[62] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E. Haque, Lingjia Tang, and Jason Mars. 2018. The Architectural Implications of Autonomous Driving: Constraints and Acceleration. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[63] Bharat B Madan, Manoj Banik, and Doina Bein. 2016. Securing unmanned autonomous systems from cyber threats. *Journal of Defense Modeling & Simulation* 16, 2 (2016), 119–135.

[64] Vicente Matellán, Jesús Balsa, F. Casado, Camino Fernández, and Francisco Javier Rodríguez Lera. 2016. Cybersecurity in Autonomous Systems: Evaluating the performance of hardening ROS. In *XVII Workshop En Agentes Físicos.*

[65] Jarrod R. Mcclean and Charles Farrar. 2013. A Preliminary Cyber-Physical Security Assessment of the Robot Operating System (ROS). In *Proceedings of SPIE.*

[66] Chandrakana Nandi and Michael D. Ernst. 2016. Automatic Trigger Generation for Rule-based Smart Homes. In *PLAS@CCS.*

[67] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V. Krishnamurthy, Edward J. M. Colbert, and Patrick D. McDaniel. 2018. IotSan: fortifying the safety of IoT systems. In *Conference on Emerging Network Experiment and Technology (CoNEXT).*

[68] NTyler Nighswander, Brent M. Ledvina, Jonathan Diamond, Robert Brumley, and David Brumley. 2012. GPS software attacks. In *ACM Conference on Computer and Communications Security (CCS).*

[69] Davide Quarta, Marcello Pogliani, Mario Polino, Federico Maggi, Andrea Maria Zanchettin, and Stefano Zanero. 2017. An Experimental Security Analysis of an Industrial Robot Controller. In *IEEE Symposium on Security and Privacy (S&P).*

[70] Davide Quarta, Marcello Pogliani, Mario Polino, Andrea M. Zanchettin, and Stefano Zanero. 2017. *Rogue robots: Testing the limits of an industrial robot's security.* Technical Report. Politecnico di Milano.

[71] Nils Miro Rodday, Ricardo de Oliveira Schmidt, and Aiko Pras. 2016. Exploring security vulnerabilities of unmanned aerial vehicles. In *IEEE/IFIP Network Operations and Management Symposium (NOMS).*

[72] Seong-Hun Seo, Byung-Hyun Lee, Sung-Hyuck Im, and Gyu-In Jee. 2015. Effect of Spoofing on Unmanned Aerial Vehicle using Counterfeited GPS Signal. *Journal of Positioning, Navigation, and Timing* 4, 2 (2015), 57–65.

[73] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. 2016. Reinforcement Learning for Autonomous Driving. In *NIPS Workshop on Learning, Inference and Control of Multi-Agent Systems.*

[74] Hocheol Shin, Dohyun Kim, Yujin Kwon, and Yongdae Kim. 2017. Illusion and Dazzle: Adversarial Optical Channel Exploits Against Lidars for Automotive Applications. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES).*

[75] Yasser Shoukry, Paul D. Martin, Paulo Tabuada, and Mani B. Srivastava. 2013. Non-invasive Spoofing Attacks for Anti-lock Braking Systems. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES).*

[76] Siciliano, Bruno, and Oussama Khatib. 2016. *Springer handbook of robotics.* Springer, Secaucus, NJ, USA: Sprinter-Verlag New York, Inc.

[77] Yunmok Son, Hocheol Shin, Dongkwan Kim, Young-Seok Park, Juhwan Noh, Kibum Choi, Jungwoo Choi, and Yongdae Kim. 2015. Rocking Drones with Intentional Sound Noise on Gyroscopic Sensors. In *USENIX Security Symposium (USENIX Security 15).*

[78] Sebastian Thrun, Wolfram Burgard, and Diter Fox. 2005. *Probabilistic Robotics.* The MIT Press.

[79] Nils Ole Tippenhauer, Christina Pöpper, Kasper Bonne Rasmussen, and Srdjan Capkun. 2011. On the requirements for successful GPS spoofing attacks. In *ACM Conference on Computer and Communications Security (CCS).*

[80] Russell Toris, Craig A. Shue, and Sonia Chernova. 2014. Message authentication codes for secure remote non-native client connections to ROS enabled robots. In *International Conference on Technologies for Practical Robot Applications (TePRA).*

[81] Timothy Trippel, Ofir Weisse, Wenyuan Xu, Peter Honeyman, and Kevin Fu. 2017. WALNUT: Waging Doubt on the Integrity of MEMS Accelerometers with Acoustic Injection Attacks. In *EuroS&P.*

[82] Yazhou Tu, Zhiqiang Lin, Insup Lee, and Xiali Hei. 2018. Injected and Delivered: Fabricating Implicit Control over Actuation Systems by Spoofing Inertial Sensors. In *USENIX Security Symposium (USENIX Security 18).*

[83] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A. Gunter. 2019. Charting the Attack Surface of Trigger-Action IoT Platforms. In *ACM Conference on Computer and Communications Security (CCS).*

[84] Jon S Warner and Roger G Johnston. 2002. A simple demonstration that the global positioning system (GPS) is vulnerable to spoofing. *Journal of Security Administration* 25, 2 (2002), 19–27.

[85] Ruffin White, Henrik I. Christensen, and Morgan Quigley. 2016. SROS: Securing ROS over the wire, in the graph, and through the kernel. In *CoRR abs1611.07060.*

[86] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. 2015. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the Internet-of-Things. In *HotNets.*

[87] Kexiong (Curtis) Zeng, Shinan Liu, Yuanchao Shu, Dong Wang, Haoyu Li, Yanzhi Dou, Gang Wang, and Yaling Yang. 2018. All Your GPS Are Belong To Us: Towards Stealthy Manipulation of Road Navigation Systems. In *USENIX Security Symposium (USENIX Security 18).*

[88] Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenbury, Shan Lu, and Blase Ur. 2019. AutoTap: synthesizing and repairing trigger-action programs using LTL properties. In *International Conference on Software Engineering (ICSE).*

A APP INTERACTION GRAPH

Figure 15 shows a complete interaction graph of home app. Gray ellipses denote the function nodes; while rectangles represent topics. Each two nodes are connected through topics. We use black arrows to denote these interactions. The interactions of GR-ST and GR-MT are marked with blue and red, receptively. We also use purple rectangles with/without diagonal stripes to denote MSR event-related and MSR action-related topics. The RSR image-related nodes is depicted in yellow ellipses.

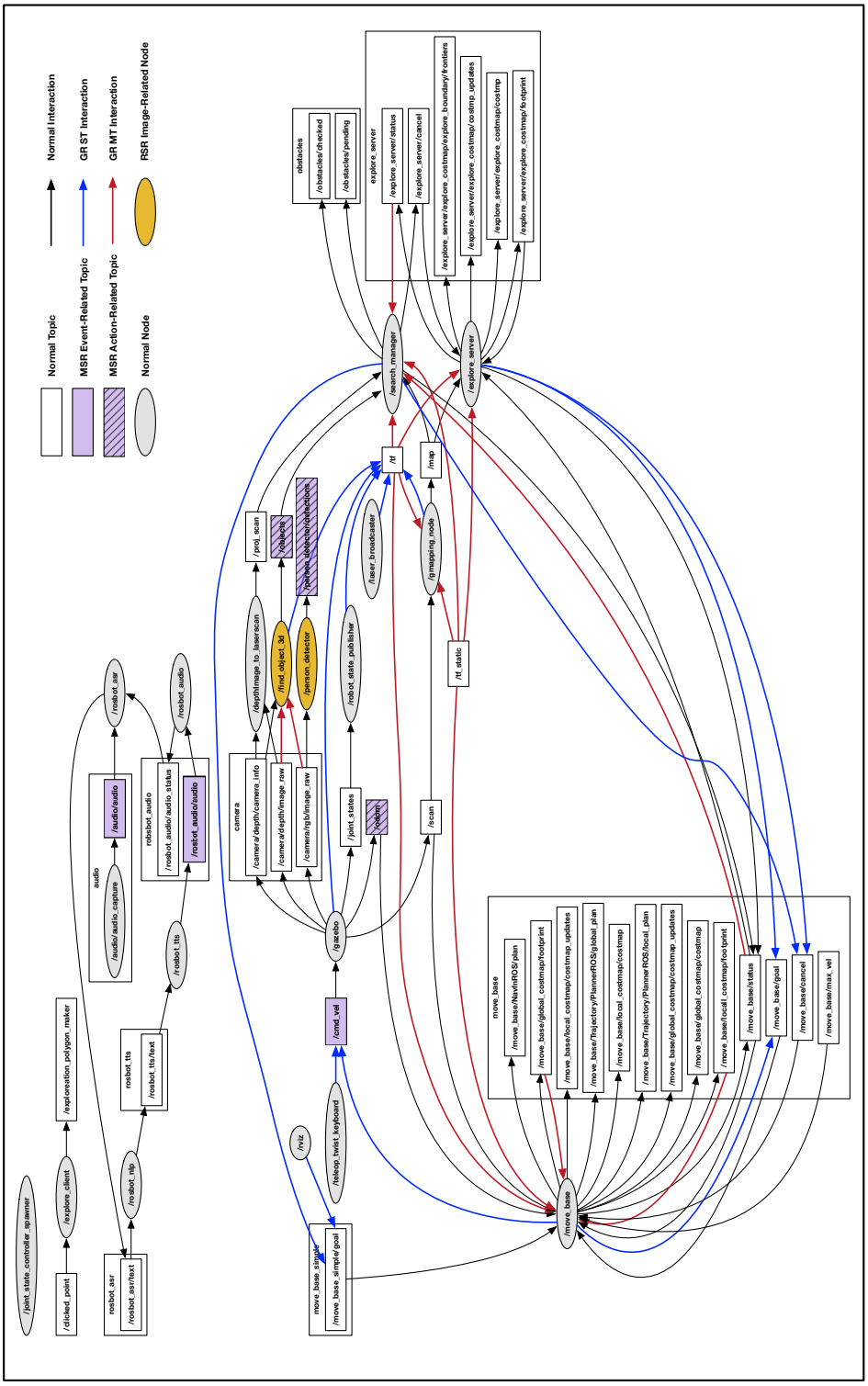


Figure 15: The Interaction Graph of Robot Applications in Home Scenario. The subscriptions of visualization node (i.e. /rviz) and log node (i.e. /rosnode) are deleted in the figure.