This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3348772

Function Interaction Risks in Robot Apps: Analysis and Policy-based Solution

Yuan Xu, Yungang Bao, *Member, IEEE,* Sa Wang, *Member, IEEE,* and Tianwei Zhang

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. We introduce a comprehensive graph to model the function interactions in robot apps by analyzing 3,100 packages from the Robot Operating System (ROS) platform. From this graph, we identify and categorize three types of interaction risks. Second, we propose novel methodologies 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 showed our methodologies can correctly identify and mitigate all potential risks.

Index Terms—Function interaction, risk identification, risk mitigation, robot apps

1 Introduction

Robotic vehicles, such as automated vehicles, drones and self-driving cars are assisting humans with any dangerous or tedious jobs. In order to adapt to different conditions, 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.

As functions become more complex and their number increases, many companies encapsulate these functions as interfaces to ease the development of robot apps for their products, such as Ford Open XC [1], Dji Onboard SDK [2], UR Application Builder [3]. Developers can then use these functions to create new apps. Another solution is public platforms, 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) [4], 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 [2], PR2 humanoid [5] and ABB manipulator [6].

However, these functions can be the Achilles' Heel of robot apps, threatening the safety of robot operations. This hazard derives from two observations. One is code sharing without any security check. Different from other well-developed app stores (e.g., mobile devices [7], PCs [8], IoT [9]), 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. The other is the interaction feature among different function nodes in the robot apps 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 [10]. For instance, Chrysler Corporation recalled 1.4 million vehicles in 2015 due to a software vulnerability in its Uconnect dashboard computers [11]. An adversary could exploit it to hack into a jeep remotely and take over the dashboard functions.

To ensure the safety of robot apps, we need to answer these two questions: *What potential risks and security incidents can a malicious interaction bring? How can we mitigate malicious interactions inside the apps?* Unfortunately, there are currently few studies focusing on the interactions in robot apps. Security analysis of interactions in IoT systems have been explored [12]–[14]. As robot apps have more complex and distinct features, it is hard to apply the above methods to the robot ecosystem, as discussed in § 7.

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 introduce a comprehensive interaction graph to model all the interactions in robot apps. Although numerous apps have been implemented for various robot devices and tasks, there are still no systematic summaries about the characteristics of function node interactions. To achieve this, we implement a rule-based tool to automatically analyze 3100 packages from the ROS platform, categorize them into 17 types and build a graph to cover all possible interactions. We also select

Y. Xu and T. Zhang is with School of Computer Science and Engineering, Nanyang Technological University. Singapore 639798, Singapore. (Corresponding author: T. Zhang.)
Y. Bao and S. Wang are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, University of Chinese Academy of Sciences, Beijing 100190, China and Peng Cheng Laboratory, Shenzhen, China. (e-mail: xuyuan@ict.ac.cn).

110 popular robot apps from the ROS platform to verify our interaction graph. This model lays a foundation for our risk analysis and mitigation in this paper, and also aims to enhance people's understanding about the characteristics of robot apps for other purposes in the future.

Second, 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 the 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.

Third, we introduce a list of novel methodologies to detect and mitigate risks from suspicious interactions in robot apps. The core of our methodologies 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 risk. At runtime, end users can observe the high-risk interactions and enforce the desired policy to the corresponding coordination node if a risk occurs.

We conduct extensive experiments to evaluate the effectiveness of our methodologies. (1) We select 110 robot apps from the ROS platform, covering 24 robots of 4 types. We 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 [15] and Apollo [16]), which are widely adopted in the autonomous vehicle industry. We successfully identify 198 high-risk interactions in these 4 apps, and mitigate them promptly and effectively.

This paper is the extension of our previous work [17]. Compared to the previous work, we made the following extensions and changes: (1) We designed a novel tool to analyze 3100 data packets in the ROS platform and model the function interactions in the robot apps. (2) We verified that the tool can identify all potential risk interactions in robot apps. (3) We ported our methodology from the ROS 1 version to the DDS-based ROS 2 version, thereby enhancing its applicability and practicality.

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 [18],

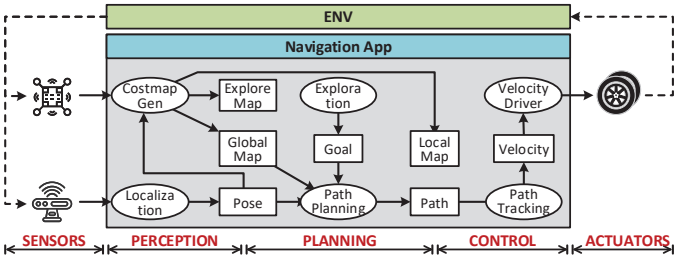


Fig. 1: An example of the navigation app.

[19]: (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 share common robot states (squares). Robot states are defined as the collection of all aspects and knowledge of the device that can impact future behaviors [20], 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 1 in the Appendix File gives an interaction graph for a real-world home-based robot app.

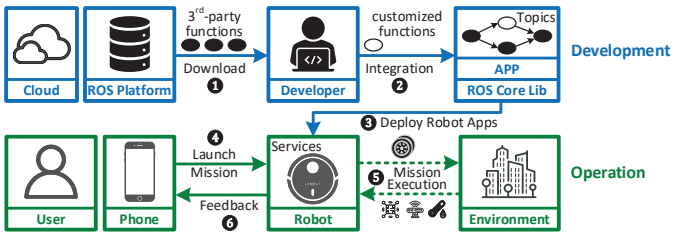


Fig. 2: The lifecycle of robot app development (blue parts) and operation (green parts).

2.2 Robot App Platform

In robotics, the most popular app platform is Robot Operating System (ROS) [4]. Both the research community and industry widely adopt ROS as the foundation or the testbed for their apps, such as Dji Matrice 200 drone [2], PR2 humanoid [5] and ABB manipulator [6]. 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.

Development and operation of robot apps. Figure 2 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).

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 [21].

This threat model is drawn from four observations. First, the ROS platform is open for everyone to upload and share their code repos without any security check. 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 [22]: "*third-party components releasing process create additional security threats (third-party components may be compromised during their distribution)*". Second, the quality of third-party function code is not guaranteed. A lot of functions in the ROS platform 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 [21], [23]. By inspecting the latest commit logs in the Robot Vulnerability Database [24], 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. Finally, this threat model is widely adopted in prior works regarding ROS security [25], [26].

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 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 [21], [27] and mitigate vulnerabilities from networks [28], sensors [29]–[36], actuators [37], [38] and controllers [39] are orthogonal to our work.

3 Function Interaction Analysis

In this section, we aim to draw an interaction graph to model all the functions in the ROS platform and their communications. Up to the date of writing, the ROS platform contains 941 repos with around 3,100 packages. A function is typically composed of multiple packages, while a repo is usually developed for one specific function. Hence, we first implement an automatic tool to categorize these repos based on the function type they can achieve[1] (§ 3.1). Then we build an interaction graph based on this categorization (§ 3.2).

3.1 Categorization of Robot Functions

We first build an automatic tool to perform large-scale analysis on all the repos from the ROS platform. Past

1. There are a few complex repos which can realize more than one functions. Such a repo commonly comprises multiple sub-packages. Our tool splits them into sub-repos by analyzing the package.xml file in each sub-package. Each sub-repo focus on one function.

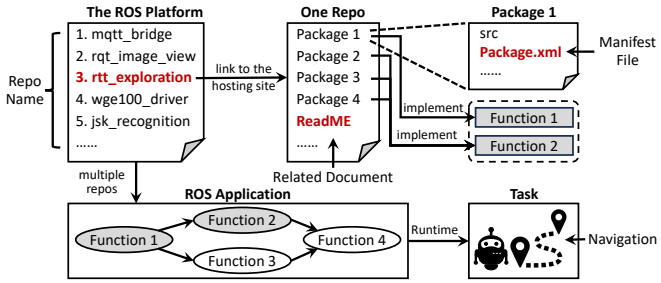


Fig. 3: The relationship among the app, repo, package, function and task.

works adopted Natural Language Processing and code dependency clustering techniques to analyze smartphone and home-based IoT apps [40]–[46]. However, it is challenging to apply these techniques to the ROS apps. First, a large portion of repos have poor code quality and the descriptions are not well documented, which can hardly reflect the characteristics of the functions. According to our analysis, 340 out of 941 repos do not provide the function descriptions in the documents, while 14 repos even use Japanese or German for the description. Second, the dependencies across repos do not provide accurate or useful information for clustering. On one hand, most repos achieve simple functions (e.g. localization, teleoperation) without importing any other repos. On the other hand, some common repos are imported by various repos without any connections. For instance, `cv_bridge` calls the OpenCV library to process images. It can be imported by the recognition, QR-based localization or vision-based mapping using function calls, rather than publish-subscribe message protocol.

Instead, we adopt the rule-based approach to analyze and categorize repos. We use the Stanford TokensRegex [47] to implement such an automatic tool. This tool includes two processes: *key information extraction* and *function classification*.

Key Information Extraction. To identify the function type of a ROS repo, three particular attributes of the repos can be inspected:

1) the *repo name*, which can directly reflect the functionality of this repo;
2) the *manifest file* (i.e. package.xml), which shows a functional brief of the repo.
3) the *related document* (i.e. README file), which presents the detailed information of this package, including function description, topics and services;

Figure 3 illustrates the interrelationships of various components within the ROS architecture. As described in § 2.2, ROS systematically maintains an inventory of indices (i.e. *repo names*), each linked to their respective source code within the hosting platform. A repository comprises single or multiple ROS packages, alongside their associated documentation, specifically, a *README file*. Each package encompasses source code and a manifest file, otherwise known as '*package.xml*'. This manifest file primarily serves the purpose of detailing version information, descriptions, and dependencies. Notably, a specific robotic function may be realized through the implementation of a single package or a combination of multiple packages, suggesting that a repository can contain more than one functionality. The developers can construct a ROS app by integrating these

TABLE 1: The successful identifications in 941 repos.

Function Type	Repo Name	Manifest File	ReadMe File	Manual	Automation Rate
Preprocessing	18	41	13	12	85.71%
Localization	17	13	3	2	94.29%
Mapping	15	11	4	1	96.77%
Recognition	21	23	4	2	96.00%
Path Planning	44	49	5	5	95.15%
Goal Planner	5	4	0	2	81.82%
Path Tracking	8	37	11	12	82.35%
Teleoperation	7	21	1	1	96.67%
Speech Generation	2	1	5	1	88.89%
Switch	3	2	1	0	100.00%
Mobile	2	26	4	6	84.21%
Manipulator	2	30	2	3	91.89%
Speaker	0	0	5	1	83.33%
Sensor	10	77	16	25	80.47%
Visualization	98	60	11	0	100.00%
Support	14	45	25	27	75.68%
Extension	23	72	18	106	51.60%
Top 14 Function Type	154	335	74	73	88.52%

functions with functions from other repositories or their customized functions. At runtime, all nodes in the app transmit data through interactions and cooperate to execute a task, such as navigation and exploration.

Listing 1 demonstrates an example of the three critical attributes sourced from the RRT exploration repository. We view the package name as an essential data point, with the remaining two key pieces of information being extracted from the manifest file and the associated documentation. Specifically, within the manifest file, we target text encapsulated within the 'description' tags as a pivotal piece of information. Regarding the related documentation, our initial step involves filtering out irrelevant distractions, such as installation commands and prerequisites outlined in the example. Subsequently, we employ the remaining repository description as another fundamental piece of information.

```
*** the repo name ***
rrt_exploration
*** the manifest file (package.xml) ***
<description>
A ROS package that implements a multi-robot RRT-
based map exploration algorithm. It also has the
image-based frontier detection that uses image
processing to extract frontier points
</description>
......
*** the related document (README) ***
It is a ROS package that implements a multi-robot
map exploration algorithm for mobile robots. It is
based on the Rapidly-Exploring Random Tree (RRT)
algorithm. It uses occupancy girds as a map repre-
sentation. The package has 5 different ROS nodes:
(1) Global RRT frontier point detector node.
(2) Local RRT frontier point detector node.
......
1. Requirements
The package has been tested on both ROS Kinetic and
ROS Indigo, it should work on other distributions
like Jade.
$ sudo apt-get install ros-kinetic-gmapping
......
2. Installation
Download the package and place it inside the /src
folder in your workspace. And then compile using
catkin_make.
......
```

Listing 1: An example of the three attributes

Function Classification. To categorize 941 repos into 17 types of robotic functions in Table 1, we first select 500 repos as the training set and assign them to 5 annotators

with rich ROS development capabilities. These annotators manually classify the function types of each repo from three particular attributes mentioned above. In the event of ambiguity, further classification was achieved via deeper analysis of dependencies and source code. To estimate the consistency of the analysis results across the 5 annotators on the 17 robotic function types, we calculated the Fleiss' kappa coefficient, which yielded a score of 0.92. We observed that inconsistencies were primarily present in repos with either insufficient documentation or those that fit into multiple types concurrently. Then we use string regular expressions to match these rules with tokens in the key information over the rest of 441 repos. Finally, we manually analyze these 441 repos to verify the correctness of our automatic classification. The matching rules for different functions are listed in Table 2. Taking the `Visualization` function as an example, the rule is to determine whether the key information contains one of two kinds of tokens. One token is the function-related words and their variants, such as visualize and simulation. The other token is the function-related tool names, such rqt and gazebo. These tools are designed to visualize the robot and different tasks. Due to the fact that the more information probably hides more inferences, we identify the function types from repo name, manifest file and related documents successively, which means the matching process is finished if it succeeds in one piece of key information. The rest of repos that cannot be detected automatically will be analyzed manually.

Table 1 shows the results of our automated function classification. The automation rate is the ratio of the successfully identified number to the sum of repos in this function. We can observe that the automation rate of most function types is more than 80% except the `Support` and `Extension` functions. Fortunately, the repos in the `Visualization`, `Support` and `Extension` belong to the *Others* domain, they are independent of the function nodes in the interaction graph. Thus, considering the repos in the first 14 function types, the automation rate of function nodes can reach 88.52%. Among these successfully automated classified repos, the accuracy of our classification can achieve 99.82%.

Results. In our evaluation, this tool can successfully identify the function types of 88.52% ROS repos. The rest repos (11.48%) are some corner cases. For example, the language used in some repos is too vague or has no description at all. For these cases, we analyzed manually, with very minor effort. We believe our tool can adapt to future ROS repos as well. It is worth noting that a repo may contain multiple tokens at the same time, which means that the repo belongs to multiple function types. For example, the turtlebot repo contains both `Extension` token (urdf) and `teleoperation` token (teleop). Thus, the sum of the repo numbers in Table 2 (i.e. 1135) is larger than the number of our target repos (i.e. 941).

Table 2 illustrates our analysis results about all function types of repos under the ROS1 kinetic version in the platform: 941 repos are split into 1135 ones as some repos implement two or more functions. Then they are categorized into 17 function types in five domains.

The first three domains include main functions for computing the robot tasks. In the *Perception* domain, the `Preprocessing` function has the largest proportion, which converts the raw sensory data or calibrates multiple data to the desired representation. Then the processed data are transmitted to the `Localization` and `Mapping` functions to form the knowledge of navigation, or transmitted to the `Recognition` function to generate the corresponding information. In the *Planning* domain, functions are used to parse a high-level task to a set of low-level actions based on the knowledge from the environment. For example, the `Path Planning` function receives the coordinate of the destination and computes a collision-free path to reach that position. The `Goal Planner` function encapsulates each action into a basic unit and defines the corresponding logic execution sequence. The planned actions are then forwarded to the functions in the *Control* domain to drive the actuators. The actuators can be commonly classified into three categories: wheels or rotors in mobile robots, manipulator and speaker. The first two actuators are driven by the `Path Tracking` function, which generates adaptive velocities to control these robots navigating in the real world. The `Speech Generation` function creates audios for the speaker to respond to users' requests. This function also includes packages from many cloud providers. The `switcher` function is used to switch on/off certain actuators for some specific tasks.

In addition to the three main domains, we also identify two more categories. In the *Drivers* domain, the functions allow the robot to interface with various actuators. The *Others* domain has the largest number of repos. Specifically, the `Visualization` function provides a GUI plugin for users to control the robot. The `Support` function builds a bridge between ROS and many third-party platforms, making the robot apps compatible with AI frameworks, mobile apps, and cloud services. The `Extension` function provides the wrappers of core libraries in ROS to ease the robot app development.

3.2 Building an Interaction Graph

With the above categorization, we analyze the message types between different repos, and generate an abstract interaction graph, as shown in Figure 4. From the figure, we can observe that functions in *Perception*, *Planning* and *Control* domains constitute all the computational nodes (the gray ellipse) in the interaction graph[2].

Key Information Extraction. Our approach to identifying all interactions within a variety of robot apps begins with recognizing the messages transmitted between different functional nodes. As discussed in § 2.1, function nodes share robot states through direct interactions. The robot states are generally estimated values the function node computes based on the sensory data. We use a solid arrow to represent the direct interaction between two nodes, which is implemented in a topic pattern. Specifically, two functions are connected if the source node publishes message types which are subscribed by the destination node. The message type published/subscribed by adjacent nodes is labeled above the arrow. For example, the published message type of the `Path Planning` function is 'nav_msgs/Path',

2. According to the sensor types supported by ROS [48], we further split the `Recognition` function into several recognition subfunctions, e.g. temperature/illuminance/humidity recognition.

TABLE 2: Categorization of repos in the ROS platform.

Domain	Function Type	# of Repos	# Rule	Example Repos
Perception (17.6%)	Preprocessing	84 (7.4%)	(calibrat | preprocess | vision | image | laser | gps | imu | point cloud | depth cloud)/w+	lidar_camera_calibration
	Localization	35 (3.1%)	localiz/w+ | slam	slam_karto
	Mapping	31 (2.7%)	map/w+ | slam	homer_mapping
	Recognition	50 (4.4%)	((detect | recogni)/w+ | identification) (face | lane | temperature | luminescence | humidity | ...)	jsk_recognition
Planning (10%)	Path Planning	103 (9.1%)	nav/w+ | moveit | explor/w+ | global plan/w+	robot_navigation
	Goal Planner	11 (1%)	plan/w+ (behavior | goal | mission)/w+	behaviortree_planner
Control (10%)	Path Tracking	68 (6%)	local plan/w+ | track/w+	teb_local_planner_tutorials
	Teleoperation	30 (2.6%)	teleop/w+	joy_teleop
	Speech Generation	9 (0.8%)	talker | tts | speaker | generate speech	homer_tts
	Switcher	6 (0.5%)	iot | internet of thing	iot_bridge
Drivers (18.4%)	Mobile	38 (3.3%)	(driver | hardware interface) (mobile | wheel)	ackermann_controller
	Manipulator	37 (3.3%)	(driver | hardware interface) (manip | grasp)/w+	agile_grasp
	Speaker	6 (0.5%)	(driver | hardware interface) (speech | audio | tts)	xbot_talker
	Sensors	128 (11.3%)	(driver | hardware interface) (lidar | camera | radar | imu | sensors ... (exclude above three types))	xsens_driver
Others (44%)	Visualization	169 (14.9%)	view | (visual | rqt | simula | rviz)/w+	rqt_reconfigure
	Support	111 (9.8%)	(bridge | dependency | library | plugin | wrapper)/w+	aws_ros1_common
	Extension	219 (19.3%)	(ros | msgs | description | urdf | tools)/w+	ros_pytest

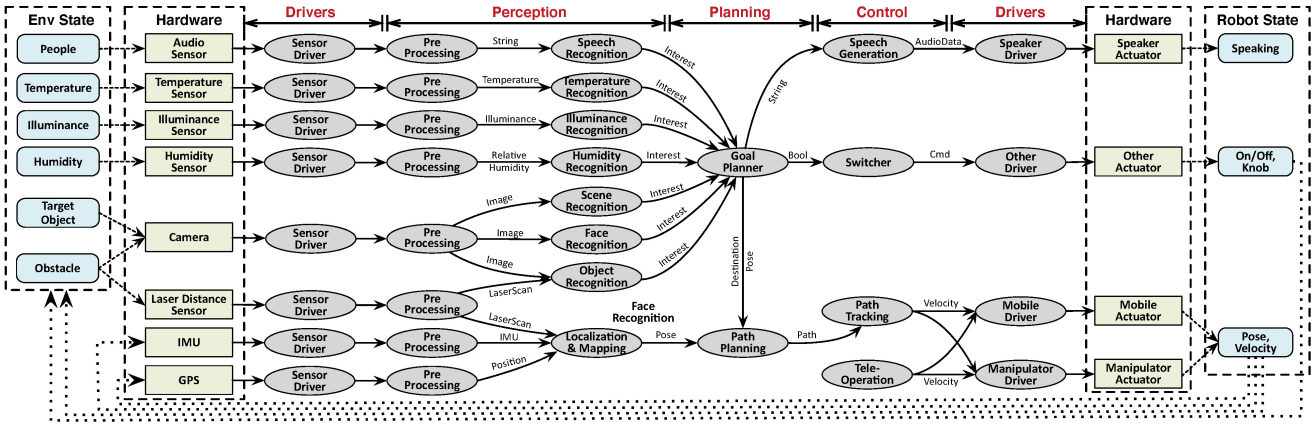


Fig. 4: The interaction graph of robot functions.

which is the same as the subscribed message type of `Path Tracking` function. Thus, a connection appears from the `Path Planning` to `Path Tracking`.

In order to extract shared messages across diverse repositories of the ROS platform, we undertook a thorough analysis of two primary sources: README files and source code. As depicted in Listing 2, the majority of the standard README files disclose subscription and publication topic names (highlighted in gray) and their associated message types (indicated in blue) relevant to the repository. Simultaneously, the number of message types defined by ROS is finite, encompassing 32 basic message types [49] (e.g., ColorRGBA, Time) as determined by its core, and an additional 86 internal state types [50] (e.g., Path, imu). A matching process between these type names and their corresponding characters in the README files enables the extraction of all related topics. Subsequent categorization is achieved by associating these with the relevant 'Publish' or 'Subscribe' keywords. Moreover, ROS offers some specific function interfaces for subscription and publishing. Identifying these interface names within the source code can also locate corresponding topics and related operations. Service extraction is also similar to the topics, which can also be obtained through related interfaces and keywords.

*** **the related document (README)** ***

4.2.2. Subscribed Topics

• The map (Topic name is defined by the ~map_topic parameter) (nav_msgs/OccupancyGrid).

44.2.3. Published Topics

• detected_points (geometry_msgs/PointStamped Message): The topic on which the node publishes detected frontier points.

.....

*** **the source code** ***

```
ros::Subscriber sub= nh.subscribe(map_topic, 100,
mapCallBack);
void mapCallBack(const gnav_msgs::OccupancyGrid::
ConstPtr& msg)
......
ros::Publisher targetspub = nh.advertise<
geometry_msgs::PointStamped>("/detected_points", 10);
......
```

Listing 2: An example of the two sources

Interaction Graph. By connecting all these function nodes with identified message types, we introduce an interaction graph of robot functions to reveal such a complete closed-loop robot operation in Figure 4. Specifically, function nodes in the *Perception* domain receive physical environmental information from sensors and convert it to robot internal estimated states (e.g. pose). Function nodes in the *Planning* domain generate long-term plans (e.g. path) based on robot's knowledge. All data flows converge to the function nodes in the *Control* domain, which output action commands (e.g. velocity) to control related actuators. Then the actuators can alter the surrounding environment, and force the function nodes in the *Perception* to repeat the above procedure.

In addition, from Figure 4, we can also observe that function nodes also have indirect interactions via the physical environment. The action commands generated from the function nodes in the *Control* domain can alter the robot's surrounding environment, which will also affect some function nodes as they need to re-estimate the latest robot states and determine the actions at the next moment. We use dotted arrows to represent such indirect interactions caused by

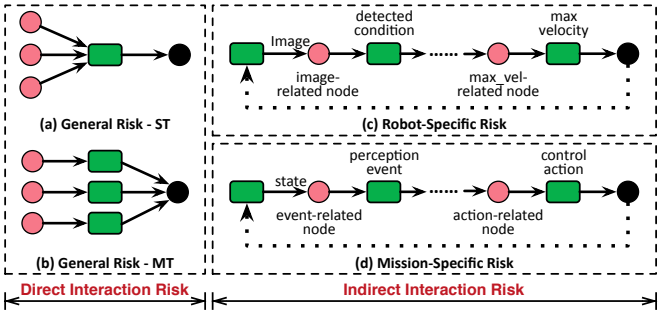


Fig. 5: Three types of interaction risk.

the changes of environmental contexts. For example, after the `Path Tracking` node generates velocity and drives the robot to a new context, the obstacle's position relative to the robot has also changed. Then the function nodes in *Perception* need to update the robot states (e.g. map and pose). If an obstacle is added to an unknown map, the `Path Planning` node may need to re-plan a new path.

4 Risk Analysis

We analyze safety risks caused by malicious function nodes and interactions. We classify these risks into three categories (Figure 5 and Table 4). We describe how each risk can incur unexpected behaviors to threaten the robot's safety.

4.1 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 5a). (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 5b).

4.2 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 [51], [52]. 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. This paper only focuses 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 5c 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.

4.3 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 5d), 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 3 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 productivity, such as manipulators and autonomous ground vehicles (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 an 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 3: 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.

4.4 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 a threat can cause GR as well.

For instance, an autonomous vehicle is navigating the 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 on 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 3).

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.

Algorithm 1: 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

foreach *topics* $t_j \in T$ **do**
  if *num*$(N_j^p) > 1$ **then**
    $RN_{gr}^{ST} \leftarrow \left\{N_j^p\right\}$;
  if *('max_vel'* $\in t_j.name) \wedge (t_j.type ==$ *'std_msgs/Float64')* **then**
    $RN_{rsr}^{max} \leftarrow \left\{N_j^p\right\}$;
  foreach *string* $s_n \in$ *EVENT_MSG_TYPE* **do**
    if $(s_n \in t_j.type) \vee$ *('detect'* $\in t_j.name)$ **then**
      $RN_{msr}^{event} \leftarrow \left\{N_j^p\right\}$;
  foreach *string* $s_n \in$ *ACTION_MSG_TYPE* **do**
    if $s_n \in t_j.type \vee$ *('goal'* $\in t_j.name)$ **then**
      $RN_{msr}^{action} \leftarrow \left\{N_j^p\right\}$;
foreach *node* $n_i \in N$ **do**
  sort node's subscriptions T_i^s by $T_i^s.type$;
  foreach *subscription* $s_k \in T_i^s$ **do**
    if $s_k.type == s_{k+1}.type$ **then**
      $RN_{gr}^{MT} \leftarrow \{n_i\}$;
  foreach *subscription* $s_k \in T_i^s$ **do**
    if $s_k.type ==$ *'sensor_msgs/Image'* **then**
      foreach *publication* $p_m \in T_i^p$ **do**
        foreach *string* $s_n \in$ *RECOG_TOPIC_NAME* **do**
          if $s_n \in p_m.name$ **then**
            $RN_{rsr}^{image} \leftarrow \{n_i\}$;

5 Mitigation Methodology

We present a novel methodology to mitigate the malicious function interactions. The core of our solution is a set of *coordination nodes* (§ 5.1) and *security policies* (§ 5.2), as summarized in Table 4.

5.1 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. Note that coordination nodes are not derivatives of third-party libraries, and their underlying logic is maintained relatively straightforward to preclude the possibility of embedded malicious code. Indeed, the scenario wherein an adversary circumvents security and seizes control of a node via hacking or other means, may indeed pose a risk. We recognize this as a legitimate concern; however, such a scenario is orthogonal to our current research.

5.1.1 Coordination Node Deployment

We design three types of coordination nodes, to mitigate three types of risks respectively (Figure 6).

General Risk Coordination Node (GRCN). This node is inserted between the high-risk nodes and their successor

TABLE 4: 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.

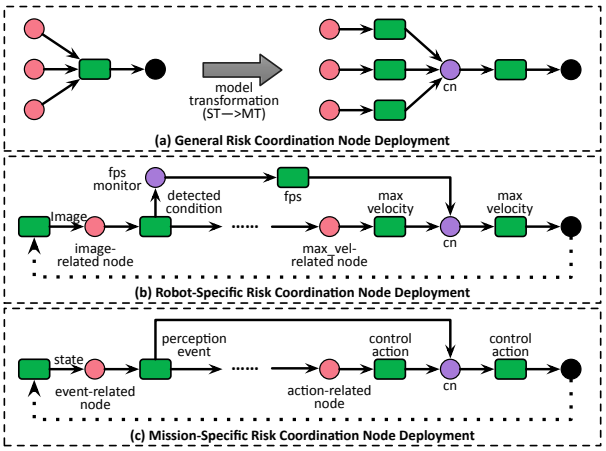


Fig. 6: Three types of coordination nodes (purple circles).

node (Figure 6a). 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 6b). 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 6c). 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.

5.1.2 Potential Risk Discovery

To locate the deployment position of each coordination node, we propose Algorithm 1 to identify all potential interactions and related high-risk nodes. The procedure commences with simulating the lifecycle of the application offline, thus auto-generating the interaction graph. The graph is then traversed, wherein all high-risk function nodes are automatically identified using Algorithm 1. The identification for each type of high-risk nodes involves the following set of rules:

TABLE 5: 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 6: 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.

GR Rule: Nodes that publish topics with an indegree greater than 1 are identified and denoted as RN_{gr}^{st} with single topics (Lines 1-3). Nodes that subscribe to more than one topic of the same message type are regarded as RN_{gr}^{mt} with multiple topics (Lines 12-16).

RSR Rule: To identify image-related node RN_{rsr}^{image} and max_vel-related node RN_{rsr}^{max}, the topic name and type of each subscribed or published message is checked (Lines 4-5, 17-22). We search the keywords (e.g., 'detect', 'people', 'face') in the RECOG_TOPIC_NAME string list. Evaluations in § 6 indicate this keyword searching can effectively identify the RSR nodes.

MSR Rule: For the identification of event-related node RN_{msr}^{event} and action-related node RN_{msr}^{action}, the message type of each topic is checked (Lines 6-11). The check is made against the EVENT_MSG_TYPE or ACTION_MSG_TYPE lists as the message types commonly follow standard ROS naming conventions [53]. Detailed lists of EVENT_MSG_TYPE and ACTION_MSG_TYPE are shown in Table 5 and Table 6.

5.2 Security Policies

To mitigate the malicious interactions in an app, each type of coordination nodes implements a set of policies. Table 4 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 the 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-related 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 inexperienced 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 the 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. These two parameters are configured based on specific contexts and tasks. For example, in scenarios where a robot navigates an area densely populated with people and a speed constraint is in effect, users can manually set the upper limit of the robot's speed. In this way, even if malicious nodes tamper with the speed through high-risk interactions, the robot can be guaranteed to navigate at a safe speed. Table 4 shows the role of end users and developers for each policy (the "Executor" column).

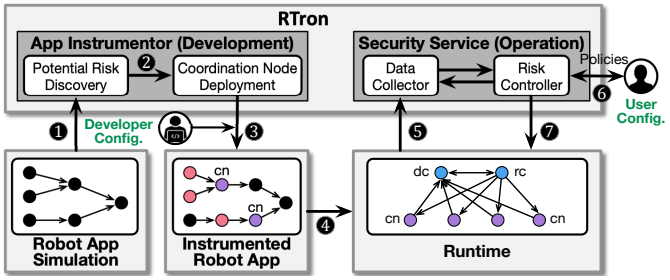


Fig. 7: RTRON system overview.

5.3 Methodology Implementation

We design RTRON, a novel end-to-end system implementing methodologies mentioned above. RTRON enables the developer to integrate necessary coordination nodes into a potentially vulnerable robot app without altering the original function nodes and set up specific security policies. Then the end user can safely launch the patched app on the robot, and configure other policies before the task starts. Figure 7 illustrates the overview of RTRON, which comprises two main components: *app instrumentor* and *security service*.

App Instrumentor. This module aims to identify all potential risks from the source code of the target app and automatically deploys coordination nodes to capture events and actions from high-risk function nodes. Specifically, the *Potential Risk Discovery* submodule first simulates the target app's lifecycle and automatically constructs the interaction graph offline (❶). By using Algorithm 1, it traverses all function nodes (black circles in Figure 7) within the graph and generates the information identified from three types of risks (❷). The collected data is then employed to set up the coordination nodes. All these nodes would be automatically deployed in the app by the *Coordination Node Deployment* submodule after the developer configures a set of topics and parameters for each node (❸). Topics specify the data transition between the potential node and coordination node and parameters provide the policy choices and related options of each policy to end users. Meanwhile, the developers also check the details of the risks, select the optional policies for GRCN and configure related parameters.

Figure 8(a) shows a GRCN instance. The GRCN continuously tracks velocity data from three risky nodes: `Navigation Control`, `Tele-operation` and `Safe Control`. The data transmission of each node is denoted as flow1, flow2, and flow3. After launching the app, the developer have the option to select the Priority_Queue policy and assign the highest priority to flow3 from `Safe Control`, signifying that its velocity action should always be prioritized. However, if the coordination node fails to receive the corresponding actions within a specified timeout duration (e.g., 0.2s), it will proceed to transmit the velocity action of flow2, which holds the second-highest priority.

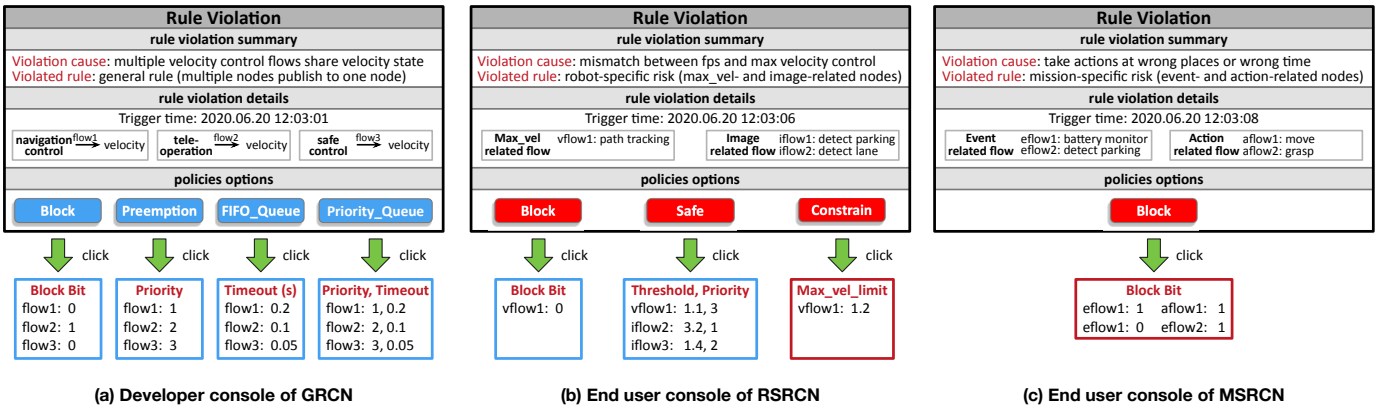


Fig. 8: Developer and end user console of each risk in RTRON. The red solid rectangle denotes a button for the end users. The blue/red box represents policy-related configuration parameters for the developers/end users.

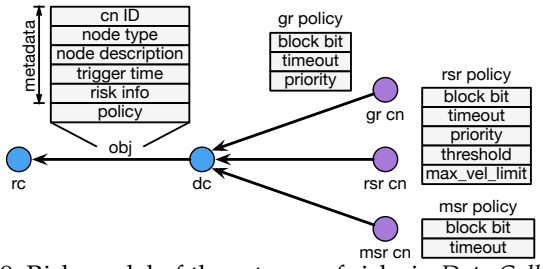


Fig. 9: Risk model of three types of risks in *Data Collector*.

Security Service. This module aims to visualize and mitigate the risks of malicious interactions at runtime. As the robot executes the task within its environment (❹), all coordination nodes in the instrumented app continuously transmit their data to the *Data Collector* submodule (❺). This information is stored as a risk model, comprising metadata and a collection of policy parameters. As shown in Figure 9, the metadata records basic information of a coordination node, including its ID, node type, node description, trigger time and risk information. This metadata is then used by the *Risk Controller* submodule to visualize risk data and provide interfaces for end users to enforce policies to each coordination node (❻). Some of these policies are mandatory, while others are optional, depending on the real-world demands (e.g., task or scenario). Figure 8 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. Once all policies are configured by the end user, the *Risk Controller* will send the user-defined parameters to each coordination node to safeguard the robot (❼).

Taking RSRCN as an example (Figure 8(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.

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

1) 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.

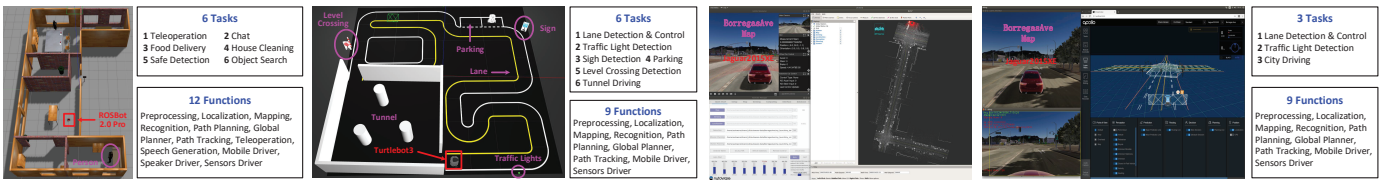
2) Risk Mitigation. Risks are mitigated in both the development stage and operation stage. As shown in Figures 8(a) and (b), the developer can choose the GRCN policy (blue button), and customize GRCN and part of RSRCN parameters for each policy (blue square). In the operation stage, the end users can 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 our interaction graph model all the interactions in robot apps? (§ 6.1)
- Can we effectively detect three types of interaction risks? What is the relationship between the interaction risks and task characteristics in each robot app? (§ 6.2)
- 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? (§ 6.3)
- What is the performance overhead of RTRON? (§ 6.4)

Testbed. We study 110 open-source apps from the ROS showcase website [54], covering 24 different robots including mobile base (MB), mobile manipulator (MM), micro aerial vehicle (MAV) and humanoid robot (HR). Table 7 summarizes the categories of these apps, numbers and the



(a) Home (b) Autorace (c) Autoware (d) Baidu Apollo

Fig. 10: Four simulated scenarios in the Gazebo/LGSVL.

applicable robot types. In addition, we also perform analysis of more complex apps (Figure 10):

- *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 [55] to develop one home app (Figure 10a).
- *AutoRace scenario [56]*: 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 [57] which can realize all six tasks in the autorace scenario (Figure 10b).
- *Autonomous driving scenario*: we consider two mainstream self-driving apps: Autoware [15] and Apollo [16], 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 10c and 10d).

It is worth noting that the 110 open-source apps drawn from the ROS showcase website are basic code offerings by various open-source robot companies. These apps tend to comprise several function nodes designed to perform a single and straightforward task such as area navigation or exploration. In stark contrast, real-world scenarios frequently present intricate circumstances where robots are required to respond to numerous events within a specific scenario - from avoiding obstacles to obeying traffic signals and even performing overtaking. Consequently, to assess our method's efficacy more accurately and practically, we elected to focus on these 4 complex apps, which more closely mirror real-world situations.

Experimental Setup. Since this paper focuses on the software risks in robot apps, we mainly use simulation to validate our solution. We choose the Gazebo simulator [58] and ROS Kinetic in the home and autorace scenarios and the LGSVL simulator [59] with ROS Indigo for Apollo 3.5. We use Rviz [60] to visualize 3D information from both the simulator and robot apps.

6.1 App Analysis

Table 7 summarizes the categories of these apps, numbers and the applicable robot types. All these apps can be illustrated using our interaction model, as shown in the fourth column. We describe the top five robot apps in the ROS platform that are most commonly used, according to Table 7.

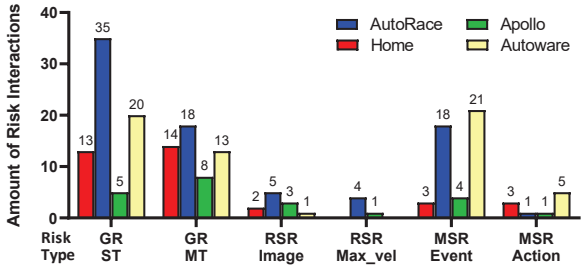


Fig. 11: Numbers of high-risk nodes in four robot apps.

Remote control. This type of apps is designed to control the robot remotely from smartphones, joysticks or keyboards. It uses `Teleoperration` to receive the signals from the remote controller and `Mobile/Manipulator Driver` to transfer these signals to each actuator's control command.

2D/3D mapping. These apps aim to create a 2D/3D map of an unknown environment through remote control. End users use `Teleoperration` to move the robot to explore the unknown zones. During the exploration, `Preprocessing` sends structural sensory data to `Mapping` for map creation.

Navigation. This type of apps instructs a robot to navigate through an obstacle-filled known environment and reach a specified destination. These apps use `Localization` to estimate the robot's position, and `Path Planning` to compute a collision-free path from its position to the destination. Then, `Path Tracking` is called to follow the path until the robot achieves the goal or the mission fails.

SLAM. These apps can be regarded as the combination of Mapping and Navigation. To reach an arbitrary destination in an obstacle-filled unknown environment, the apps use `Mapping` and `Localization` simultaneously to transfer the unknown map to a known one and locate its position.

Face/Person Detection. These apps receive images from cameras (`Preprocessing`) and apply the OpenCV face/person detector based on an Adaboost cascade of Haar features/HOG (`Recognition`). They publish regions of interests (ROIs) of the detection and a debug image, showing the processed image with the ROIs that is likely to contain faces or persons.

6.2 Risk Identification

We successfully identify 198 risk interactions in the four target apps. The verification process for risks involves a two-step process. We commence by generating an interaction graph for each complex app, as depicted in Figure

TABLE 7: Analysis of open-source robot apps from the ROS showcase website [54].

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

TABLE 8: 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

10. Following this, a manual examination of each high-risk interaction identified by our method is undertaken to confirm their accuracy, while simultaneously ensuring no risks have been overlooked. Figure 11 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. Table 8 gives examples of the identified high-risk nodes for each type in Home-based and AutoRace app. Texts marked in red are for risk identification in our system.

6.3 Risk Mitigation

CN Analysis. We use 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, we check the GR information of "Pub Node" and deploy the GRCN between high-risk nodes and their pub nodes. For RSRs, since RSRCNs directly publish velocity messages to the `Mobile Driver` function, the number of RSRCN is always 1. The subscriptions of RSRCN are related to the number of image-related nodes and max_vel-related nodes. Besides, as described in § 5.1, 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 9: Numbers of CNs in four complex robot apps.

Scenario	GRCN			RSR		MSR
	Perception	Planning	Control	FMN	CN	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 9 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. We implement 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 10, the messages of interacted topics in perception are related to the sensory information (e.g. images) or 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

TABLE 10: 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

TABLE 11: Processing time of potential risk discovery.

Application	Node Number	Topic Number	Processing Time (s)		
			GR	RSR	MSR
Teleoperation [61]	4	17	0.114	0.113	0.057
Voice Interaction [62]	6	7	0.035	0.035	0.011
Mapping [63]	6	25	0.308	0.299	0.152
Navigation [64]	8	63	0.764	0.727	0.498
Exploration [65]	10	84	1.12	1.086	0.753
Home	21	125	3.121	3.199	1.927
AutoRace [56]	25	112	4.075	4.049	2.105
Apollo [16]	21	39	0.631	0.606	0.306
Autoware [15]	38	218	2.945	2.931	1.747

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

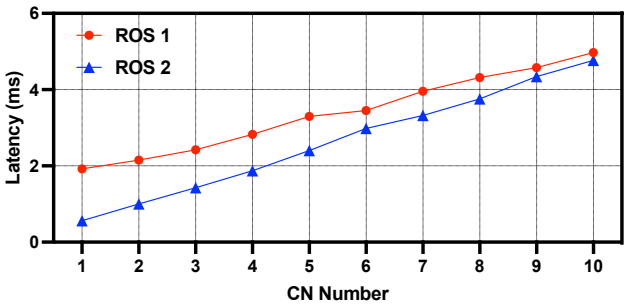


Fig. 12: Overhead of CNs in an end-to-end data flow.

(`/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.

Effectiveness Evaluation. We assume that a node in a high-risk interaction is manipulated by an attacker and sends malicious messages to its successor nodes. In the absence of coordination nodes, the attack success rate is 100%. But when we deploy the coordination node, because the topic is renamed, all malicious messages will be intercepted by the coordination node, and decisions will be made through the policy. We found that our method can also achieve 100% success rate in preventing such attacks that exploit interaction when the policy is chosen correctly. For the detailed process, please refer to the video: https://youtu.be/fZeLJEZkfec.

6.4 Performance Overhead

Offline Overhead Evaluation. We evaluated the processing time in the RTRON's risk discovery phase, specifically the high-risk node identification in various robot apps. The performance results, as outlined in Table 11 , were obtained from nine unique apps, each with different quantities of topics and nodes. Experimental data was collected from multiple iterations (n=20) to compute an average latency. The risk identification process, which operates offline, imposes a minimal overhead. It was also observed that the number of nodes and topics influences the processing duration, as the risk discovery process depends on the traversal of either nodes or topics (Algorithm 1). The GR and RSR discovery processes both involve two iterations, while the MSR discovery process requires only one iteration of topics. Consequently, the processing time required for GR and RSR discovery is almost identical, and typically exceeds that of MSR. A noteworthy exception to this pattern was observed with the autorace application, which despite having fewer nodes and topics than the home application, demonstrated the greatest processing time. This anomaly can be attributed to a higher incidence of high-risk GR interactions (9), necessitating additional operations (i.e. matching related topic type and name) during node iteration.

Runtime overhead. Runtime overhead can be attributed to two primary factors: the coordination nodes and the 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. Similar to IoT policy enforcement systems [66], [67],

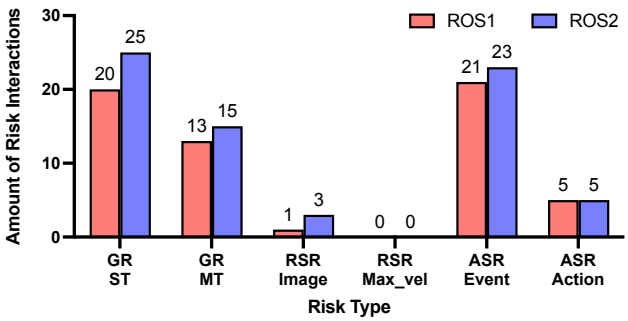


Fig. 13: Numbers of high-risk nodes in ROS1-based and ROS2-based Autoware app.

we ignored this process's overhead since user intervention is required solely during mission launch or scenario alterations. Conversely, coordination nodes, distributed among function nodes within the robotic app, can potentially extend end-to-end latency from perception to control stages. Notwithstanding, although there are dozens of nodes in a typical robot app, these nodes work in a parallel multi-flow mode. Commonly, 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 12, the latency induced by these 10 nodes is around 5ms (red circle line). This is trivial even for the autonomous driving app with the strongest real-time constraint: according to the industry standards published by Mobileye [68] and design specifications from Udacity [69], 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.

6.5 Transferability Analysis

We have upgraded our methodology from the ROS 1 to the DDS-based ROS2 [70]. The DDS framework is designed to provide higher efficiency and reliability, low latency, and scalability, as well as configurable quality of service (QoS) parameters. These elements are universally desirable in robotic platforms and adopted by most commercial robots.

Effectiveness Analysis. To assess the effectiveness of our approach in the ROS2 setting, we separately analyzed the ROS1 and ROS2-based open-source autonomous driving app - Autoware. Figure 13 illustrates the high-risk nodes identified by RTRON in these two settings. It is evident that our method shows more potential risks in the ROS2-based Autoware app, due to the integration of more functions in ROS2, resulting in a higher number of nodes and interactions compared to the ROS1-based version. We did not evaluate other apps like Baidu Apollo because the latest versions of these apps utilize custom-developed middleware. Nonetheless, since these middleware, similar to ROS2, rely on DDS distributed architecture, our approach is applicable to these frameworks as well. We have successfully ported our methodology to a DDS-based autonomous driving system provided by our partner, Desay Automotive.

Performance Analysis. Since the offline processing time of the potential risk discovery module is associated with Algorithm 1, rather than the system, we focus on the evaluation of RTRON's runtime overhead in both ROS1 and ROS2 environments. As Figure 12 shows, despite an increase in coordination nodes to 10, the overall end-to-end latency still remains below 5ms (blue triangle line). It is worth noting that unlike the ROS1 with a centralized architecture, DDS-based ROS2 employs a distributed architecture. In such a context, all nodes are independent and do not need a master node to coordinate. Therefore, in scenarios with a small number of coordination nodes, the latency observed in the ROS2 context is notably lower than that in the ROS1 context.

7 Related Works

Robotic Security. Existing research on robotic security has mainly focused on traditional security issues in robot systems, e.g., network communication [27], denial-of-service attacks [21] and software vulnerabilities [39], [71], [72]. In addition, adversaries can also spoof the sensory data ([29]–[34], [36]), fake the actuator signals [38], or tamper with the micro-controller input [39].

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.

Interaction Risk Mitigation. Prior work studied the interaction risks in IoT apps [12]–[14]. Users adopt operation rules following the "If-This-Then-That" (IFTTT) trigger-action programming paradigm [73], [74] 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. For instance, a heater control app might activate a heater at a preset time, whereas another temperature control app opens the window upon detecting temperatures above a predefined threshold. This unexpected interaction would cause a potential risk of thief entry. The current approach focuses on enforcing an "allow/block" policy to enhance IoT security after identifying such high-risk interactions.

However, applying these methods to robot apps presents challenges. As noted in Table 4, risk mitigation policies must be tailored to specific interaction risks. Consider a scenario with an autonomous vehicle: if its speed does not match with the frame rate processing, a direct blocking of the speed control, as per the IoT approach, would lead to abrupt braking on highways, and then a rear-end collision. This difference arises from the distinct natures of the IoT and RV ecosystems. In IoT, users link sensors and actuators via rules. These rules use environmental factors (time, humidity, etc.) to create indirect interactions among various IoT devices. Such interactions are mostly independent and operate within a fixed closed-loop space, so blocking one interaction has little effect on most other interactions. Conversely, RVs operate in dynamic environments with interdependent interactions. As Figure 4 shows, all interactions will finally converge into some unified control flows, ensuring RV safety. Such a structure makes blocking any non-redundant interaction possible with errors in the final control flow, which in turn indirectly crashes the entire RV. Furthermore, risks in RVs derive not only from indirect interactions, i.e., RSR and MSR, but also from direct ones, i.e., GR, given the holistic decision-making of onboard controllers. This is markedly different from the decentralized nature of IoT.

Thus, addressing RV interaction risks is more complex, requiring concurrent efforts from developers and users in enforcing risk mitigation policies.

8 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.

ACKNOWLEDGEMENT

This work was supported in part by Singapore MoE AcRF Tier 1 RG108/19 (S), NTU-Desay Research Program 2018-0980, 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).

References

[1] "Openxc platform," http://openxcplatform.com/, 2020.
[2] "Dji onboard sdk," https://developer.dji.com/onboard-sdk/, 2020.
[3] "Application builder," https://www.universal-robots.com/builder/, 2020.
[4] "Open source robot operating system," http://www.ros.org/, 2019.
[5] "Ros pr2 package," http://wiki.ros.org/Robots/PR2/, 2020.
[6] "Ros abb package," http://wiki.ros.org/abb/, 2020.
[7] "Google play," https://play.google.com/store/, 2020.
[8] "Ubuntu appstore," https://ubuntu.com/blog/tag/appstore/, 2020.
[9] "Samsung smartthings," https://www.smartthings.com/, 2020.
[10] N. DeMarinis, S. Tellex, V. P. Kemerlis, G. D. Konidaris, and R. Fonseca, "Scanning the internet for ros: A view of security in robotics research," in *International Conference on Robotics and Automation (ICRA)*, 2019.
[11] "After jeep hack, chrysler recalls 1.4m vehicles for bug-fix," https://www.wired.com/2015/07/jeep-hack-chrysler-recalls-1-4m-vehicles-bug-fix/, 2015.
[12] H. Chi, Q. Zeng, X. Du, and J. Yu, "Cross-app interference threats in smart homes: Categorization, detection and handling," in *CoRR abs/1808.02125*, 2018.
[13] L. Zhang, W. He, J. Martinez, N. Brackenbury, S. Lu, and B. Ur, "Autotap: synthesizing and repairing trigger-action programs using ltl properties," in *International Conference on Software Engineering (ICSE)*, 2019.
[14] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter, "Charting the attack surface of trigger-action iot platforms," in *ACM Conference on Computer and Communications Security (CCS)*, 2019.
[15] "The autoware.ai project," https://github.com/Autoware-AI/autoware.ai, 2020.
[16] "Baidu apollo," https://github.com/ApolloAuto/apollo, 2020.
[17] Y. Xu, T. Zhang, and Y. Bao, "Analysis and mitigation of function interaction risks in robot apps," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2021.
[18] Siciliano, Bruno, and O. Khatib, *Springer handbook of robotics*. Secaucus, NJ, USA: Sprinter-Verlag New York, Inc.: Springer, 2016.
[19] Y. Xu, T. Zhang, J. Han, S. Wang, and Y. Bao, "Towards practical cloud offloading for low-cost ground vehicle workloads," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021.
[20] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. The MIT Press, 2005.
[21] B. Dieber, B. Breiling, S. Taurer, S. Kacianka, S. Rass, and P. Schartner, "Security for the robot operating system," *IEEE Trans. Robotics and Autonomous Systems*, vol. 98, pp. 192–203, 2017.
[22] "Ros 2 robotic systems threat model," https://design.ros2.org/articles/ros2_threat_model.html, 2020.
[23] J. R. Mcclean and C. Farrar, "A preliminary cyber-physical security assessment of the robot operating system (ros)," in *Proceedings of SPIE*, 2013.
[24] "Robot vulnerability database (rvd)," https://github.com/aliasrobotics/RVD/, 2020.
[25] D. K. Hong, J. Kloosterman, Y. Jin, Y. Cao, Q. A. Chen, S. A. Mahlke, and Z. M. Mao, "Avguardian: Detecting and mitigating publish-subscribe overprivilege for autonomous vehicle systems," in *European Symposium on Security and Privacy (EuroS&P)*, 2020.
[26] B. Breiling, B. Dieber, and P. Schartner, "Secure communication for the robot operating system," in *IEEE Systems Conference (SysCon)*, 2017.
[27] B. Dieber, S. Kacianka, S. Rass, and P. Schartner, "Application-level security for ros-based applications," in *International Conference on Intelligent RObots and Systems (IROS)*, 2016.
[28] J. Chen, Z. Feng, J.-Y. Wen, B. Liu, and L. Sha, "A container-based dos attack-resilient control framework for real-time uav systems," in *Design, Automation, and Test in Europe (DATE)*, 2019.
[29] N. O. Tippenhauer, C. Pöpper, K. B. Rasmussen, and S. Capkun, "On the requirements for successful gps spoofing attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2011.
[30] N. Nighswander, B. M. Ledvina, J. Diamond, R. Brumley, and D. Brumley, "Gps software attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2012.
[31] K. C. Zeng, S. Liu, Y. Shu, D. Wang, H. Li, Y. Dou, G. Wang, and Y. Yang, "All your gps are belong to us: Towards stealthy manipulation of road navigation systems," in *USENIX Security Symposium (USENIX Security 18)*, 2018.
[32] H. Shin, D. Kim, Y. Kwon, and Y. Kim, "Illusion and dazzle: Adversarial optical channel exploits against lidars for automotive applications," in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2017.
[33] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao, "Adversarial sensor attack on lidar-based perception in autonomous driving," in *ACM Conference on Computer and Communications Security (CCS)*, 2019.
[34] D. Davidson, H. Wu, R. Jellinek, V. Singh, and T. Ristenpart, "Controlling uavs with sensor input spoofing attacks," in *Workshop on Offensive Technologies (WOOT)*, 2016.
[35] Y. Tu, Z. Lin, I. Lee, and X. Hei, "Injected and delivered: Fabricating implicit control over actuation systems by spoofing inertial sensors," in *USENIX Security Symposium (USENIX Security 18)*, 2018.
[36] Y. Shoukry, P. D. Martin, P. Tabuada, and M. B. Srivastava, "Non-invasive spoofing attacks for anti-lock braking systems," in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2013.
[37] F. Fei, Z. Tu, R. Yu, T. Kim, X. Zhang, D. Xu, and X. Deng, "Cross-layer retrofitting of uavs against cyber-physical attacks," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2018.
[38] H. Choi, W.-C. Lee, Y. Aafer, F. Fei, Z. Tu, X. Zhang, D. Xu, and X. Xinyan, "Detecting attacks against robotic vehicles: A control invariant approach," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.
[39] D. Quarta, M. Pogliani, M. Polino, F. Maggi, A. M. Zanchettin, and S. Zanero, "An experimental security analysis of an industrial robot controller," in *IEEE Symposium on Security and Privacy (S&P)*, 2017.
[40] X. Pan, Y. Cao, X. Du, B. He, G. Fang, R. Shao, and Y. Chen, "Flowcog: Context-aware semantics extraction and analysis of information flow leaks in android apps," in *USENIX Security Symposium (USENIX Security 18)*, 2018.

[41] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[42] R. Xu, H. Saïdi, and R. J. Anderson, "Aurasium: Practical policy enforcement for android applications," in *USENIX Security Symposium (USENIX Security 12)*, 2012.

[43] X. yong Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, location, disease and more: inferring your secrets from android public resources," in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[44] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, "Flowfence: Practical data protection for emerging iot application frameworks," in *USENIX Security Symposium (USENIX Security 16)*, 2016.

[45] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "Iotpot: Analysing the rise of iot compromises," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[46] X. yong Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[47] "Stanford tokensregex," https://nlp.stanford.edu/software/tokensregex.html, 2020.

[48] "Sensors supported by ros." http://wiki.ros.org/Sensors/, 2020.

[49] "Standard ros messages," http://wiki.ros.org/std_msgs, 2023.

[50] "Common ros messages," http://wiki.ros.org/common_msgs, 2023.

[51] B. Boroujerdian, H. Genc, S. Krishnan, W. Cui, A. Faust, and V. J. Reddi, "Mavbench: Micro aerial vehicle benchmarking," in *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[52] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[53] "Ros messages," http://wiki.ros.org/Messages/, 2020.

[54] "Robots that you can use with ros." https://robots.ros.org/, 2020.

[55] "Rosbot 2.0 pro," https://store.husarion.com/collections/dev-kits/products/rosbot-pro/, 2020.

[56] "Turtlebot3 autorace," https://emanual.robotis.com/docs/en/platform/turtlebot3/autonomous_driving, 2020.

[57] "Turtlebot3," https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/, 2020.

[58] "Gazebo 3d robot simulator." http://gazebosim.org/, 2020.

[59] "Lgsvl simulator." https://www.lgsvlsimulator.com/, 2020.

[60] "Rviz 3d visualization tool for ros." https://www.stereolabs.com/docs/ros/rviz/, 2020.

[61] "Rosbot teleoperation app," https://husarion.com/tutorials/ros-tutorials/3-simple-kinematics-for-mobile-robot/, 2020.

[62] "Xiaoqiang voice interaction app," https://community.bwbot.org/topic/492/, 2020.

[63] "Rosbot slam app," https://husarion.com/tutorials/ros-tutorials/6-slam-navigation/, 2020.

[64] "Rosbot navigation app," https://husarion.com/tutorials/ros-tutorials/7-path-planning/, 2020.

[65] "Rosbot exploration app," https://husarion.com/tutorials/ros-tutorials/8-unknown-environment-exploration/, 2020.

[66] W. Ding and H. Hu, "On the safety of iot device physical interaction control," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[67] Z. B. Celik, G. Tan, and P. D. McDaniel, "Iotguard: Dynamic enforcement of security and safety policy in commodity iot," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[68] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Reinforcement learning for autonomous driving," in *NIPS Workshop on Learning, Inference and Control of Multi-Agent Systems*, 2016.

[69] "An open source self-driving car," https://www.udacity.com/self-driving-car/, 2020.

[70] "Ros2 document," https://docs.ros.org/en/galactic/index.html, 2023.

[71] C. G. L. Krishna and R. R. Murphy, "A review on cybersecurity vulnerabilities for unmanned aerial vehicles," in *IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, 2017.

[72] T. Kim, C. H. Kim, J. Rhee, F. Fei, Z. Tu, G. Walkup, X. Zhang, X. Deng, and D. Xu, "Rvfuzzer: Finding input validation bugs in robotic vehicles through control-guided testing," in *USENIX Security Symposium (USENIX Security 19)*, 2019.

[73] C. Nandi and M. D. Ernst, "Automatic trigger generation for rule-based smart homes," in *PLAS@CCS*, 2016.

[74] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu, "Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things," in *HotNets*, 2015.

Yuan Xu is a research fellow in School of Computer Science and Engineering, at Nanyang Technological University. His research interests include secure robot system, performance optimization, and physical attacks. He received his Ph.D. degree in computer engineering from Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2021. He received his master degree in Software Engineering from University of Sciences and Technology of China in 2013.

Yungang Bao is a professor of the State Key Laboratory of Computer Architecture, Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS). He is the director of Research Center for Advanced Computer System (ACS). Yungang received his BS degree in computer science from Nanjing University in 2003, and his PhD degree in computer engineering from ICT, CAS in 2008. During 2010-2012, he did postdoc research in Department of Computer Science, Princeton University, working with Prof. Kai Li on the PARSEC project.

Sa Wang is an associate professor at Institute of Computing Technology, Chinese Academy of Sciences. He got his Ph.D. degree in January 2016 in Institute of Software, Chinese Academy of Sciences, advised by Prof. Tao Huang. He received his bachelor degree in Computer Science from University of Sciences and Technology of China in 2009. His research fields mainly include cloud computing, operating systems, virtualization and performance models.

Tianwei Zhang is an assistant professor in School of Computer Science and Engineering, at Nanyang Technological University. His research focuses on computer system security. He is particularly interested in security threats and defenses in machine learning systems, autonomous systems, computer architecture and distributed systems. He received his Bachelor's degree at Peking University in 2011, and the Ph.D degree in at Princeton University in 2017.