Automating ROS2 Security Policies Extraction through Static Analysis

Giacomo Zanatta[1], Gianluca Caiazza[1], Pietro Ferrara[1], Luca Negrini[1], Ruffin White[2]

Abstract— **Cybersecurity in mission-critical robotic applications is a necessity to scale deployments securely. ROS2 builds upon DDS-Security specs in ROS Client Library (RCL) to implement its security features. Utilizing SROS2, developers have access to a set of utilities to help set up security in a way RCL can use. Through SROS2, security deployment is eased for developers. However, while access control is handled by DDS and consequently based on the SROS2-generated permission artifacts, the necessary authorization policies are manually generated by developers. This requires an entire system exercise to be sampled via live extraction and, per each node, list all the necessary Topics, Services, and Actions, which is a daunting and laborious process. Developers first have to generate tests. Then, they obtain a 'snapshot' of the system for each test. Later, these snapshots must be collected and grouped into a policy by a minimum set of rules. All this procedure is quite error-prone. This paper introduces LiSA4ROS2, a tool for automatically extract the ROS2 computational graph via static analysis to derive a minimal correct configuration for ROS2 security policies. Our approach relies on the abstract interpretation theory to statically overapproximate all possible executions to extract a minimal and complete configuration per node. We evaluate our approach with minimal examples covering all the main communication patterns in ROS2 tutorials and all publicly available real-world ROS2 Python systems extracted from GitHub. The results of the minimal examples show that LiSA4ROS2 precisely supports all the main communication patterns. The extensive evaluation underlines that our prototype implementation of the analysis in LiSA4ROS2 is already able to precisely analyze 66% of existing repositories, automatically producing detailed computational graphs and access policies. All the results of the analysis, as well as a Docker artifact to reproduce them, are publicly available.**

I. Introduction

Nowadays, robotic systems, and in general, Cyber-Physical Systems (CPS), are pervasive and ubiquitous. With the spread of those devices outside of laboratories and controlled environments, such as autonomous cars and medical teleoperated robots, we have to carefully evaluate how system misbehavior could compromise humans and environmental safety [1]. Robotic frameworks were not designed initially with security in mind. However, as they started to be adopted into products and used in mission-critical programs, more attention was drawn to security issues [2].

Robot Operating System (ROS) [3], and its successor ROS2 [4] represent the *de facto* framework for robotic development. In recent years, we have observed how the interest in security in the ROS community has increased. From a community survey in 2022 [5], we know that 73% of the survey participants considered investing more to protect their robots from cyber threats. The same number of participants indicated that their organizations were open to investing; however, only 26% acknowledged having invested. There has been some early work on securing ROS [6], [7], [8], which are not active anymore due to the shift in security effort to ROS2 with the Secure ROS2 project (SROS2) [9]. Similarly to ROS2, other modern robotic and cyber-physical system protocols, such as Eclipse Zenoh[1], and Internet of Things (IoT) solutions (e.g., Apple Homekit, Google Home, Matter Alliance, etc.) leverage their complex distributed architecture on loosely coupled graph representation. Through this abstraction, we can significantly simplify the development effort and enhance code portability through different versions and technology solutions.

Currently, ROS2 builds upon Data Distribution Service (DDS) Security (DDS-Security) [2] specs in ROS Client Library (RCL) for implementing its security features. Utilizing SROS2, developers have access to a set of utilities to help set up security in a way RCL can use. Through SROS2 utilities, security deployment is eased for developers who do not have to manually manage Certificate Authorities (CAs), Nodes' identity artifacts, and governance files, providing streamlined mechanisms as well to translate ROS2 '*security terms*' in low-level DDS permissions. At the time of writing, there is an active effort in the ROS community to support alternative middleware solutions to complement Secure DDS[3]. The ROS2 core team has examined currently available middleware alternatives and identified security as one of the top requirements priority [10] with support to Encryption, Authentication, and Access Control as "*Must*" - as intended in RFC 2119 - to be included. SROS2 currently supports all the properties mentioned above, but developers, to correctly implement cybersecurity in their products, should follow some necessary steps as proposed in the SROS2 DevSecOps model [9]. The six-step methodology includes the following: (A) Modeling, (B) Authentication, (C) Authorization, (D) Generalization, (E) Deployment, and (F) Monitoring. On the one hand, steps B to E are covered in SROS2 tools by leveraging DDS. On the other hand, points A and F, as part of the never-ending loop of continuous Monitoring and Mitigation, require the developer to interact and audit the system manually. Our work is focused on the Modeling phase, where developers have to create a set of tests to exercise the application to live-extract fully, per node, the

[1]Giacomo, Gianluca, Pietro, and Luca are with Ca' Foscari University of Venice, Italy `giacomo.zanatta, gianluca.caiazza, pietro.ferrara, luca.negrini @unive.it`

[2]Ruffin with White Robotics `rwhitema@ucsd.edu`

[1]https://zenoh.io/

[2]https://www.omg.org/spec/DDS-SECURITY

[3]https://discourse.ros.org/t/investigation-into-alternative-middleware-solutions/32642

list of all the necessary *Topics*, *Services*, and *Actions*. We argue that exhaustively exercising a system by tests is a daunting and limited process (i.e., execution path). Each test results in a *snapshot* of the system, which later has to be merged with the other to build the Node's security policy. Merging is a complex process as it requires verifying that the identified set is minimal to respect the Principle of Least Privilege (PoLP), which limits access rights to only what is strictly required and correct with the Node's resources. Devising effective access control policies can be challenging and error-prone, especially when performed manually. In this article, we enhance the current approach proposed in the Modeling phase, introducing *LiSA4ROS2* [11], a tool for automatically extracting the security policy from the source code via abstract interpretation. A sound overapproximation (that is, considering all possible executions of the program) of the ROS2 computational graph is extracted, and a correct configuration for ROS2 security policies is provided. We implemented our approach in the prototype LiSA4ROS2 as an extension of the Library for Static Analysis (LiSA)[4] [12], [13]. Starting from a ROS2 Python application, LiSA4ROS2 produces the program's computational graph and the XML access policy specification. We tested our tool on two distinct sets of applications. First, we analyzed all ROS2 tutorial examples, explaining how different robotic nodes can communicate. In all these examples, LiSA4ROS2 produced exact computational graphs and access policies. We then analyzed all publicly available GitHub repositories that contain ROS2 Python programs. This extensive analysis, consisting of almost 700 repositories and more than 5000 Python files, shows that LiSA4ROS2 can precisely infer computational graphs and access policies on 66% of the repositories. The remaining 34% are not yet precisely analyzed because of some limitations in the features supported by the analysis (such as non-constant names of nodes and topics coming from external configuration files). Those features might be added to LiSA4ROS2 easily with some engineering work, but since they do not represent an interesting scientific contribution, we left them as future work. LiSA4ROS2 is a fully automatic tool that can be easily integrated with SROS2 and become part of the DevSecOps model pipeline to increase usability and security. In addition, it can be generalized to other ecosystems in robotics and CPS, where the same graph-based architecture is applicable with support to multiple languages. The corresponding artifacts, Docker images, and results are available on the GitHub repositories:

- *https://github.com/lisa-analyzer/lisa4ros2*
- *https://github.com/lisa-analyzer/lisa4ros2-fe*

The rest of the paper is organized as follows. Section II presents a motivating scenario. Section III introduces the approach developed in LiSA4ROS2. Section IV presents LiSA4ROS2 experimental evaluation. Section V examines the related works. Finally, Section VI discusses future works, and Section VII concludes.

[4]https://github.com/lisa-analyzer/lisa

II. Motivating Scenario

Statically analyzing the source code enables the identification of entities whose presence in the network cannot be (or is hard to) dynamically discover. For instance, the existence of a name may hinge on the reception of a particular message or an ad-hoc configuration, or it might be activated exclusively in specific execution environments (e.g., exclusively in production). In these scenarios, a dynamic analysis would reveal only a partial view of the communications, as it is contingent upon the timing, location, and conditions under which the analysis is executed (i.e., whether it occurs before or after a node receives the triggering message leading to the instantiation of one or more entities). Consider, for instance, the code snippet presented in Listing 1, extracted from the `solar_ros` repository[5]. This code orchestrates a conventional communication scheme among various robotic nodes. Specifically, the code defines a class named `PolyDustService` that extends a ROS Node.

```
1 class PolyDustService(Node):
2   def __init__(self):
3     super().__init__('poly_dust_service')
4     self.callback_group = MutuallyExclusiveCallbackGroup()
5     model_path = pkg_resources.resource_filename('poly_dust',
6             'sm_unet4_03.hdf5')
7     self.unet_model = Unet_Model(model_path)
8     self.srv = self.create_service(DeliverImg,
9             'poly_dust_service',
10            self.proccess_img,
11            callback_group=self.callback_group)
12    self.get_logger().info('RUNING ...')
13
14
15  def send_deliver_request(self, id):
16    deliver_client = self.create_client(DeliverImg ,
17                                         'deliver_server')
18    while not deliver_client.wait_for_service(timeout_sec=1.0):
19    self.get_logger().info('deliver_server no disponible, Esperando ...')
20    deliver_request = DeliverImg.Request()
21
22    deliver_request.photo_id = id
23
24    return deliver_client.call(deliver_request)
25
26  def recover_img_from_deliver_response(self, response):
27    img_bytes = np.array(response.photo, dtype=np.uint8)
28    return cv2.imdecode(img_bytes, _RGB)
29
30  def proccess_img(self, request, response):
31    response.photo = []
32    self.get_logger().info(f'Incoming request for poly dust server, photo
     with id {request.photo_id} asking delivery')
33    stored_photo = self.send_deliver_request(request.photo_id)
34    if stored_photo.photo == array('B'):
35      self.get_logger().info(f'Photo with id {request.photo_id} not
     available')
36          return response
37      self.get_logger().info(f'Recovered photo with id {request.photo_id},
     using model')
38      image =  self.recover_img_from_deliver_response(stored_photo)
39      image = cv2.resize(image, _image_size, interpolation = cv2.INTER_AREA)
40
41      dust_image = self.unet_model.unet_prediction(image)
42
43      self.get_logger().info(f'Resnet process eneded for photo with id {
     request.photo_id}')
44
45      reconstructed = dust_image.astype(np.uint8) * 100
46      reconstructed = array('B', reconstructed.tobytes())
47      response.photo = reconstructed
48      return response
```

Listing 1: Python source code from `solar_ros` repository

Within its constructor, a service server is instantiated (line 8) to receive a photo ID and respond with the corresponding photo. However, the retrieval of the photo involves an additional step. Initially, the photo ID is forwarded to another node through a service client that invokes a server named `deliver_server` (line 16). The instantiation of this server occurs within the `send_deliver_request` function, invoked within the callback function of the service server (line

[5]https://github.com/cocisran/solar-ros

```
/poly_dust_service
  Subscribers:

  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
  Service Servers:
    /poly_dust_service: solar_interfaces/srv/DeliverImg
    /poly_dust_service/describe_parameters: rcl_interfaces/srv/
      DescribeParameters
    /poly_dust_service/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /poly_dust_service/get_parameters: rcl_interfaces/srv/GetParameters
    /poly_dust_service/get_type_description: type_description_interfaces/srv/
      GetTypeDescription
    /poly_dust_service/list_parameters: rcl_interfaces/srv/ListParameters
    /poly_dust_service/set_parameters: rcl_interfaces/srv/SetParameters
    /poly_dust_service/set_parameters_atomically: rcl_interfaces/srv/
      SetParametersAtomically
  Service Clients:

  Action Servers:

  Action Clients:
```

(a) List info app started

```
/poly_dust_service
  Subscribers:

  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
  Service Servers:
    /poly_dust_service: solar_interfaces/srv/DeliverImg
    /poly_dust_service/describe_parameters: rcl_interfaces/srv/
      DescribeParameters
    /poly_dust_service/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /poly_dust_service/get_parameters: rcl_interfaces/srv/GetParameters
    /poly_dust_service/get_type_description: type_description_interfaces/srv/
      GetTypeDescription
    /poly_dust_service/list_parameters: rcl_interfaces/srv/ListParameters
    /poly_dust_service/set_parameters: rcl_interfaces/srv/SetParameters
    /poly_dust_service/set_parameters_atomically: rcl_interfaces/srv/
      SetParametersAtomically
  Service Clients:
    /deliver_server: solar_interfaces/srv/DeliverImg

  Action Servers:

  Action Clients:
```

(b) Output after call of the service

Fig. 1: ROS2 Output for command '*ros2 node info /poly_dust_service*'.

33). This function ultimately returns the requested image (line 48). The existence of the `deliver_server` service client is conditional upon the node serving a client to the `poly_dust_service`.

When running the program and inspecting the node using the command *ros2 node info /poly_dust_service*, the output provides a list of clients and servers, as depicted in Fig. 1a. The service client is absent since it has not yet been executed. Conversely, when requesting the `poly_dust_service` server (e.g., by executing the command '*ros2 service call /poly_dust_service solar_interfaces/srv/DeliverImg "{}"*' in a terminal) and subsequently rechecking the node's information, the client is present. This occurs because the callback has been executed at least once, as illustrated in Fig. 1b.

If the policy file is generated before the `poly_dust_service` receives at least one message using the tool provided by SROS2 (*ros2 security create policy policy.xml*), the resulting policy file, as shown in Listing 2, is incomplete.

```
<policy version="0.2.0">
  <enclaves>
    <enclave path="/">
      <profiles>
        <profile node="poly_dust_service" ns="/">
          <services reply="ALLOW">
            <service>poly_dust_service</service>
            <service>~/describe_parameters</service>
            <service>~/get_parameter_types</service>
            <service>~/get_parameters</service>
            <service>~/get_type_description</service>
            <service>~/list_parameters</service>
            <service>~/set_parameters</service>
            <service>~/set_parameters_atomically</service>
          </services>
          <topics publish="ALLOW">
            <topic>parameter_events</topic>
            <topic>rosout</topic>
          </topics>
        </profile>
      </profiles>
    </enclave>
  </enclaves>
</policy>
```

Listing 2: Policy file generated by SROS2

This nearly-minimal example highlights an inherent limitation of dynamic analyses: the necessity to execute the code across all conceivable environments, configurations, and inputs. However, practical constraints dictate that only a fraction of these scenarios can be explored within a reasonable time-frame. In contrast, a sound static analysis can consider all potential executions and identify every conceivable communication. Specifically, LiSA4ROS2 traces (i) the recorded callback at line 11, (ii) the asynchronous execution of the `process_img` method facilitated by this callback, and (iii) the subsequent execution of the service at line 33. This capability stems from LiSA4ROS2's semantic analysis, which incorporates the semantics of Python statements and the ROS2 library. The graph illustrated in Fig. 2 and the policy file listed in Listing 3 document the information inferred by LiSA4ROS2.

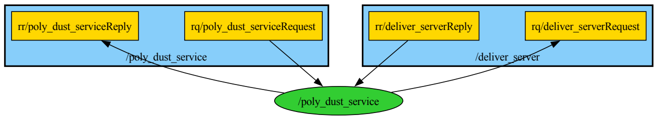


Fig. 2: LiSA4ROS2 Graph representation of the `poly_dust_service` node.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<policy version="0.2.0">
  <enclaves>
    <enclave path="/poly_dust_service">
      <profiles>
        <profile ns="/" node="poly_dust_service">
          <topics publish="ALLOW">
            <topic>/parameter_events</topic>
            <topic>/rosout</topic>
          </topics>
          <services reply="ALLOW">
            <service>/poly_dust_service/describe_parameters</service>
            <service>/poly_dust_service/get_parameter_types</service>
            <service>/poly_dust_service/set_parameters</service>
            <service>/poly_dust_service/get_parameters</service>
            <service>/poly_dust_service/get_type_description</service>
            <service>/poly_dust_service/set_parameters_atomically</service>
            <service>/poly_dust_service/list_parameters</service>
            <service>/poly_dust_service</service>
          </services>
          <services request="ALLOW">
            <service>/deliver_server</service>
          </services>
        </profile>
      </profiles>
    </enclave>
  </enclaves>
</policy>
```

Listing 3: Policy file generated by LiSA4ROS2

Nevertheless, a static approach to policy generation is not inherently foolproof. Specifically, if a name is not directly defined in the source code (e.g., sourced from a command line argument), or its constant value cannot be semantically identified (i.e., statically propagated to the method call), the analysis might fail during the generation phase.

For instance, in the code snippet in Listing 4, the name of the publisher remains unidentified since it originates from a node parameter that may be externally set or from a launch file not currently supported by LiSA4ROS2.

```
topic_name = node.get_parameter('topic_name').value

publisher = node.create_publisher(JointTrajectory,
     topic_name, 10)
```

Listing 4: ROS2 code corner case example

Additional challenges arise when a node's communication entities are defined across multiple files, invoking auxiliary functions in another Python module. Currently, LiSA4ROS2 only analyzes one file per node. The file must have an entry point such that LiSA4ROS2 can comprehend the code's execution order to extract ROS2 entity definitions accurately.

III. Approach

Figure 3 reports the architecture of our approach. The core of our analysis relies on the Library for Static Analysis (LiSA), an analysis engine that works on a generic and extensible control flow graph representation of the program to analyze. LiSA relies on the abstract interpretation theory [14], [15] to implement a sound static analysis, that is, an analysis that overapproximates all possible executions of a program without executing it.

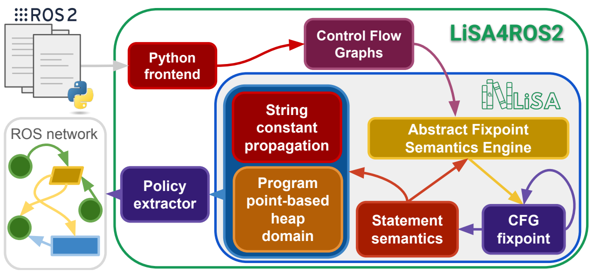


Fig. 3: LiSA4ROS2 Architecture

Our approach exploits some standard (and already implemented in LiSA) domains to approximate common information about the program. In particular, a String constant propagation domain is adopted to track constant string values [16]. Those constant values are commonly used to name the resources involved in the communication between robotic nodes. In addition, a field-sensitive program point-based heap domain [17] allows us to retrieve the internal representation of the node to track, where entities are defined and instantiated in the code. Thanks to these standard domains, which are not a contribution of this work, we can reconstruct the intercommunication of nodes. Statement semantics applies these two domains when approximating the effects of a single program statement (e.g., an assignment). Statement semantics is the basic block that allows abstract fixpoint semantics to overapproximate all the possible executions of a program on a control flow graph.

Besides the LiSA library's core, LiSA4ROS2 encompasses a Python front-end called PyLiSA. This front-end parses Python source code files and produces a control flow graph for each method. However, ROS2 Python code heavily relies upon the Python ROS library (i.e., `rclpy`). When approximating the library behaviors, practical experience shows that there is a need for manual annotation since (i) the library implements a complex logic that is hard to approximate automatically, and (ii) the code of the library is usually highly optimized. Therefore, manual annotation is required to obtain a precise analysis. For this reason, we adopted the domain-specific language SARL [18], extending it with some ROS2-specific components. In this way, we defined a sound approximation of the library runtime behaviors by manually specifying the effects of library calls on abstract analysis states.

The fact that some entities are reactive is a challenge to the analysis. This means that they execute an action (that is, a piece of code) only when some specific events occur (e.g., the ring of a timer or the reception of a message). This action is programmatically defined as a callback function. LiSA4ROS2 simulates the execution of callback functions to find if there are definitions of other network entities within them. This methodology permits the identification of various entities that are not always executed in the runtime graph. At the end of the analysis, LiSA produces an abstract entry and exit state for each statement in the program. This state overapproximates the execution state before and after a given statement. LiSA4ROS2 then processes this state to detect if a statement pushes something related to ROS in the analysis state and, in this case, stores such information in a ROS object that models a ROS network. Those ROS objects are composed of Entities (subscribers, publishers, services servers, service clients, action servers, action clients), Entity Containers (nodes), and Network Channels (topics, services, actions). An Entity has an associated Network Channel (identified by a name) and a message type. Since each node of a ROS2 application could live in a different machine, LiSA4ROS2 analyzes every program that composes the ROS2 application modularly, sharing only the ROS network object to obtain a representation of the connection between nodes.

Finally, by exploiting this information, LiSA4ROS2 extracts the ROS computational graph. Such a graph represents (i) the robotic nodes in the network (green vertexes), (ii) the topics used by the nodes to communicate (yellow vertexes), (iii) to what topics nodes publish (edges going from robotic nodes to topics), and (iv) what topics are listened to by nodes (edges going from topics to nodes). During the extraction phase, we also generate the internal topics that Services and Actions use. For example, a Service Server named `service_server` has an associated Publisher on the topic `service_serverReply` and a Subscription on the topic `service_serverRequest`. Services and Actions are identified, respectively, by cyan and violet rectangles. To generate the policy file, we convert each node of the ROS Computational Graph into an

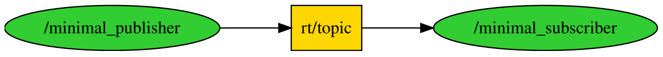


Fig. 4: Computational Graph of the minimal publisher/subscriber ROS2 tutorial example.

XML file. In addition, LiSA4ROS2 generates an HTML report per project that shows the computational graph, the number of entities extracted, and, for every node, a list of its network entities. A glimpse of such report, specific for the `minimal_subscriber` of the example seen in Fig. 4, is shown in Fig. 5: from that table, we can know that the `minimal_subscriber` has three system publishers, one system subscription, and one user-defined one. For the latter, we see that the callback function is named `listener_callback`, and it is a field of the `self` object (in that case, the node).

Motivating example

Consider again the motivating scenario introduced in Section II. At the end of LiSA analysis, among other information, the string constant propagation engine discovers that (i) the name of the robotic node passed to the constructor of the Node class at lines 4 and 8 of Listing 1 is `poly_dust_service` (reflected on the name reported on the green node of Fig. 2 and to the `poly_dust_serviceReply` and `poly_dust_serviceRequest` yellow nodes), and (ii) the server used to deliver messages is `deliver_server` as specified at line 17 (leading to the `deliver_serverReply` and `deliver_serverRequest` nodes). In addition, the program point-based heap domain enables LiSA4ROS2 to infer that (i) `poly_dust_service` receives request messages from and returns reply messages to the service created at line 8 (leading to the two edges on the left of Figure 2), and (ii) it sends request messages to and receives reply messages back from the service created at line 17 (leading instead to the two edges on the right).

IV. Evaluation

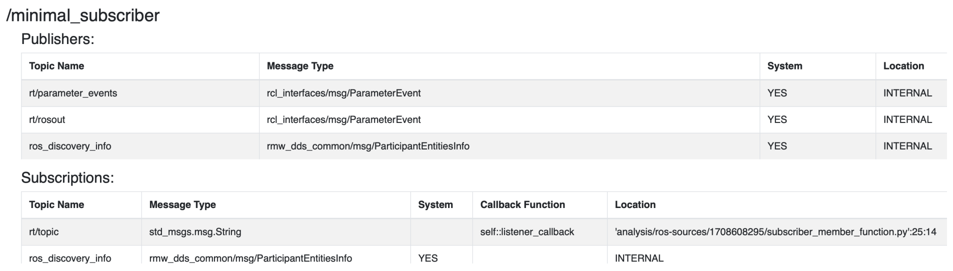
To assess the effectiveness of LiSA4ROS2, we analyzed and inspected the results on two main families of ROS2 applications: (i) some minimal examples taken from the ROS2 official tutorials, and (ii) publicly available GitHub repositories containing ROS2 Python applications. Table I presents various metrics related to a few chosen examples of both families. All the examples mentioned above and the complete analysis results are available at *https://lisa-analyzer.github.io/lisa4ros2-fe*. In addition, a Dockerized version of our tool is contained in the main repository. This can help reproduce the results reported in this section and run our analysis on other applications. In addition, upon successfully executing the analysis, as exemplified in Figure 4, the tool generates an SVG file portraying the topology of the analyzed system for visual inspection. In this representation, a green oval signifies a node, while a trapezoid represents a resource involved in communication. The visualization employs yellow rectangles for topics and light blue and violet rectangles for services and actions, respectively. The edges in the diagram depict the flow of messages. An edge from a node to a resource means the node acts as a publisher in a topic or initiates communication (e.g., serving as a services server or action client). Conversely, an edge from a resource to a node indicates that the node functions as a subscriber, service server, or action server. The analysis also provides an XML file per node representing an SROS2 policy for that node. This file defines access control policy rules that say what resources a node is authorized to use. Considering the `minimal_publisher` node of the previously mentioned example, the analysis generates the policy file listed in Listing 5.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<policy version="0.2.0">
  <enclaves>
    <enclave path="/minimal_publisher">
      <profiles>
        <profile ns="/" node="minimal_publisher">
          <topics publish="ALLOW">
            <topic>/topic</topic>
            <topic>/parameter_events</topic>
            <topic>/rosout</topic>
          </topics>
          <services reply="ALLOW">
            <service>/minimal_publisher/get_parameter_types</service>
            <service>/minimal_publisher/set_parameters_atomically</service>
            <service>/minimal_publisher/set_parameters</service>
            <service>/minimal_publisher/describe_parameters</service>
            <service>/minimal_publisher/list_parameters</service>
            <service>/minimal_publisher/get_type_description</service>
            <service>/minimal_publisher/get_parameters</service>
          </services>
        </profile>
      </profiles>
    </enclave>
  </enclaves>
</policy>
```

Listing 5: Policy file for `minimal_publisher` node.

A. *Minimal examples*

Firstly, we applied our tool to conduct an in-depth analysis of the `rclpy` examples package within ROS2, accessible at



/minimal_subscriber

Publishers:

Topic Name	Message Type	System	Location
rt/parameter_events	rcl_interfaces/msg/ParameterEvent	YES	INTERNAL
rt/rosout	rcl_interfaces/msg/ParameterEvent	YES	INTERNAL
ros_discovery_info	rmw_dds_common/msg/ParticipantEntitiesInfo	YES	INTERNAL

Subscriptions:

Topic Name	Message Type	System	Callback Function	Location
rt/topic	std_msgs.msg.String		self::listener_callback	'analysis/ros-sources/1708608295/subscriber_member_function.py':25:14
ros_discovery_info	rmw_dds_common/msg/ParticipantEntitiesInfo	YES		INTERNAL

Fig. 5: A glimpse of the HTML result for the `minimal_publisher` node

ROS2 example repo[6]. Through these examples, we aimed to showcase the tool's capabilities in handling minimal setups encompassing the most common code styles of invoking the client library API, including both local and class member functions for topic, service, and action functionality. The first four rows of Table I report metrics of some of those examples. These examples were selected since they are part of the official ROS2 documentation[7]. In addition, we included the *oldschool* version of the minimal publisher/-subscriber example just to show that our analysis can also handle different types (sometimes legacy) of node creation. We manually inspected the analyses' results on all these examples. The graphs and policies produced by LiSA4ROS2 reflected what was described and provided in the ROS2 tutorials. Therefore, we conclude that LiSA4ROS2 supports all the main types of communication of ROS2 applications, i.e., topics, services, and actions.

B. GitHub repositories

In November 2023, we downloaded a set of publicly available Python ROS2 applications from GitHub repositories. To select such applications, we downloaded a set of GitHub repositories that contain Python ROS2 applications (i.e., at least one Python file that imports the rclpy library). This comprehensive collection included a total of 682 repositories[8]. Within this dataset, we systematically extracted all Python files (launch file excluded) containing the definitions of `super.init`, `create_node`, or `Node` statements, resulting in the extraction of 5936 files. Subsequently, these files were processed through our Python front-end, with 5552 files (93.53%) successfully passing the analysis and 384 files (6.47%) encountering parsing errors — details of which are available in the repository. Further investigation involved subjecting the passing files to our control flow graph module, resulting in 5141 files (92.6%) being accurately processed, while 411 files (7.4%) encountered errors not being able to reach a fix-point in the execution. Out of the files processed, we determined that 3406 (66%) can automatically determine all the necessary information directly from the code[9]; 1735 (34%) instead are cases where the analysis failed to compute precise names (e.g., those might depend on configuration files that are currently not yet supported by LiSA4ROS2), prompting additional scrutiny and documentation[10]. The last thirteen rows of Table I report metrics of some selected examples. Those Python ROS2 applications range from a few hundred to several thousand lines of code and from less than a dozen to several hundreds of topics, services, publishers, and subscribers. The selection process was guided by considerations such as the number of elements, their interconnections, and the lines of code. Overall, they represent a set of realistic Python ROS2 applications with a relatively complex computational graph. LiSA4ROS2 precisely analyzed all these examples. For instance, `fruit_collectors` encompasses two nodes, six topics (three of which are system-related), fifteen services (with fourteen designated as system services), zero actions, and eight publishers across six systems. This comprehensive overview provides a detailed insight into the structural components and connectivity within the example systems. A notable example is the `virtuoso` repository[11]: our tool extracted 52 nodes, producing a highly connected graph. This is the biggest graph we obtained in our analysis: without considering systems entities, LiSA found 95 topics (for which we have 132 publishers and 83 subscriptions), 14 services (with 7 service servers and 11 service clients), and 1 action with 1 action client. Given the complexity of this

[6]https://github.com/ros2/examples

[7]https://docs.ros.org/en/rolling/Tutorials/Beginner-Client-Libraries.html

[8]The dataset is available from: https://doi.org/10.5281/zenodo.10817418

[9]In all these cases, the string constant propagation analysis computed precise results for node and topic names. Therefore, we are sure that the computed results are precise.

[10]The collection of results from the extracted GitHub repositories is available here: https://doi.org/10.5281/zenodo.10818685

[11]https://github.com/gt-marine-robotics-group/Virtuoso.git

TABLE I: Metrics of selected examples.

PROJECT	LOC	N	T	S	A	PUB	SUB	SS	SC	AS	AC
pubsub_minimal	44	2	4 (3)	14 (14)	0	7 (6)	3 (2)	14 (14)	0	0	0
pubsub_minimal_oldschool	38	2	4 (3)	14 (14)	0	7 (6)	3 (2)	14 (14)	0	0	0
services_minimal	41	2	3 (3)	15 (14)	0	6 (6)	2 (2)	15 (14)	1	0	0
actions_minimal	101	2	3 (3)	14 (14)	1	6 (6)	2 (2)	14 (14)	0	1	1
mechaship	849	5	8 (3)	38 (35)	0	18 (15)	13 (5)	35 (35)	3	0	0
solar-ros	243	6	4 (3)	44 (42)	0	19 (18)	7 (6)	44 (42)	3	0	0
ROS-LLM	302	4	9 (3)	30 (28)	0	19 (12)	6 (4)	30 (28)	1	0	0
Catch2023_hichewns	1000	8	30 (3)	56 (56)	0	52 (24)	34 (8)	56 (56)	0	0	0
fruit_collectors	220	2	6 (3)	15 (14)	0	8 (6)	5 (2)	15 (14)	1	0	0
spatial-topology-teleoperation	166	2	10 (3)	14 (14)	0	11 (6)	5 (2)	14 (14)	0	0	0
ProjectMarch	511	5	13 (3)	37 (35)	1	25 (15)	9 (5)	38 (35)	0	0	1
5g_drone_ROS2	882	6	12 (3)	53 (42)	0	20 (18)	15 (6)	45 (42)	11	0	0
Virtuoso	3840	52	98 (3)	378 (364)	1	288 (156)	135 (52)	371 (364)	11	0	1
eml4842_gps_nav	543	7	12 (3)	50 (49)	0	28 (21)	15 (7)	50 (49)	2	0	0
MARV-ROS	1036	7	51 (3)	49 (49)	0	59 (21)	42 (7)	49 (49)	0	0	0
module89	1947	15	28 (3)	113 (105)	0	71 (45)	33 (15)	111 (105)	7	0	0
zumopi_telemetry_system	1451	5	22 (3)	35 (35)	0	35 (15)	34 (5)	35 (35)	0	0	0

Lines Of Code (LOC), Numbers of Nodes (N), Topics (T), Services (S), Actions (A), Publishers (PUB), Subscribers (SUB), Service Servers (SS), Service Clients (SC), Action Servers (AS), Action Clients (AC). The value inside the parentheses represents system entities.

application, exposing all these communications exhaustively by runtime execution would have been infeasible.

V. Related Work

To the best of our knowledge, no tools are currently available to fulfill the specific task we intend to undertake with LiSA4ROS2. In the literature, analogous works can be found, albeit within distinct domains and applications. For example, the authors of [19] utilized static analysis for the automated extraction of database interactions in Web applications. Like LiSA, they employed static analysis to construct a Control Flow Graph (CFG), albeit in a more condensed form called an Interaction Flow Graph (IFG). Our approach stands out as more potent when compared to their method, primarily because of the restrictions they impose on their analysis, confined solely to database interactions. Instead, LiSA4ROS2 tracks communications through the publish-subscribe pattern. In contrast, our objective aligns more closely with endeavors such as [20], where the authors employed statically extracted authorization graphs in Web applications' interactions to enforce Role-Based Access Control (RBAC) on resources. They subsequently utilized their model to identify potential vulnerabilities, validate existing policies, undertake policy re-documentation, execute role mining, and simplify authorization facts. Similarly, we aim to achieve comparable results within the complexity of a multi-tiered setup, characteristic of the robotic field, where multiple nodes must access resources through publish-subscribe mechanisms that span multiple nodes and domains. Some previous work has already applied static analysis in the context of ROS, although existing approaches taken previously differ from our specific objectives. In the first iteration of ROS, notable instances include HAROS [21] and the associated analysis within the framework [22]. It is essential to note that the mentioned framework does not extend its support to ROS2, and indications suggest that such support is not anticipated[12]. Regarding ROS2, the authors of [23] focused on formally verifying the DDS component in ROS2. They provide an abstraction and a formalization of DDS in ROS2 based on probabilistic timed automata. Their goal was to verify properties such as security (i.e., no deadlock), liveness (i.e., a node can reach the send_wait), and priority (i.e., high priority node has a higher priority of sending data) were extracted for formal verification. While insightful regarding property verification, our work diverges from the specific context of our intended research. Other approaches focused on the Runtime Verification (RV) of the system. Via RV, developers can verify the behaviors of safety-critical systems that are too complex to verify formally. However, monitors are complex to program, and errors would threaten the whole system. The authors in [24] provide an overview of a formal approach to generating runtime monitors for ROS2 applications via the Formal Requirement Elicitation Tool (FRET) and the Ogma integration tool. However, their focus is on integrating ROS2 packages into larger systems, minimizing the effort that is out of the scope of our objectives. A noteworthy examination involves other RV efforts. Unfortunately, they exclusively support ROS and lack compatibility with ROS2. Examples include (i) ROSRV [25], which allows the definition of safety and security properties in a formal specification language, ensured by automatically generated monitors, (ii) ROSMonitoring [26], supporting multiple ROS distributions and being agnostic to the specification formalism, (iii) and DeRoS [27], a domain-specific language and monitoring system designed explicitly for ROS. In our pursuit of enumerating ROS2 Computational Graph resources, we focus on refining the Modeling phase outlined in the SROS2 paper [9]. As mentioned, to delve into the graph's introspection, developers can employ live extraction through the ROS2 API command line tool or leverage *scapy* for the dissection and decoding of DDS network packets [28]. We can also capitalize on the enhancements introduced in [29], extending the capabilities of `ros2_tracing` for real-time tracing of ROS2. Despite its original focus on uncovering causal links between input/output messages and indirect causal links, integrating this tool into the modeling analysis proves advantageous as a supplementary component. However, all these approaches, albeit valuable, can explore only a limited amount of all possible execution paths in the code. Constrained by the current test execution (e.g., input values and configuration environment), there exists a risk of leaving certain parts uncovered, potentially leading to misconfigurations in the access policy. Consequently, hidden resources and the potential insertion of malicious elements may persist within the system due to overly relaxed rules [30]. OSRF has developed a tool for translating the Node Interface Definition Language (NoDL) description of a ROS system into SROS2 policy [31]. However, as of the current writing, the development in that regard has encountered challenges stemming from various issues with the definition [32] and adoption of NoDL in ROS2, leading to its suspension.

[12]https://github.com/git-afsantos/haros/issues/117

VI. Future Work

As mentioned, current feature limitations hinder the complete analysis of ROS packages that either leverage external configuration files to remap ROS namespaces upon launch dynamically, split node API instantiation across multiple source files or use ROS client libraries other than `rclpy`. To further community adoption and impact of LiSA4ROS2, we would like to extend support for more diverse node source layouts, as well as transitive analysis of launch files, including configurations written via interpreted scripts (Python) or more static markup languages (XML, YAML). We also intend to add support for C++ sources using `rclcpp` via the development of an LLVM front-end for LiSA. Finally, we would like to apply Information Flow Analysis to auto-detect the declassification of private messages.

VII. Conclusion

In this paper, we introduced LiSA4ROS2, a novel tool designed to automate ROS2 computational graph extraction through static analysis. We investigated the application of formal methods and elucidated how static analysis enables us to tackle the challenge of potential errors stemming from improper policy configurations, as well as the complexities associated with manually crafting a precise policy. Furthermore, we presented an empirical assessment of the tool's efficacy, showcasing its ability to facilitate both minimal setups and intricate deployments using real-world code. This evaluation underscores the tool's practical utility and its potential to streamline security policy establishment in ROS2 environments. Lastly, we provided insights into the limitations inherent in the current version of our tool and outlined avenues for future research and development that we intend to pursue. These reflections contribute to a comprehensive understanding of the tool's capabilities and highlight opportunities for enhancement in subsequent iterations.

ACKNOWLEDGMENT

Work partially supported by SERICS (PE00000014 - CUP H73C2200089001) under the NRRP MUR program funded by the EU - NGEU, and by iNEST- Interconnected NordEst Innovation Ecosystem funded by PNRR (Mission 4.2, Investment 49 1.5) NextGeneration EU (ECS_00000043 - CUP H43C22000540006).

REFERENCES

[1] Laura Alzola Kirschgens, Irati Zamalloa Ugarte, Endika Gil-Uriarte, Aday Muñiz Rosas, and Victor Mayoral Vilches. Robot hazards: from safety to security. *CoRR*, abs/1806.06681, 2018.

[2] Víctor Mayoral-Vilches. Robot cybersecurity, a review. *International Journal of Cyber Forensics and Advanced Threat Investigations*, 0(0), 2022.

[3] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3.2, page 5. Kobe, Japan, 2009.

[4] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[5] Víctor Mayoral-Vilches. Robot cybersecurity, a review. *International Journal of Cyber Forensics and Advanced Threat Investigations*, 2022.

[6] Bernhard Dieber, Severin Kacianka, Stefan Rass, and Peter Schartner. Application-level security for ros-based applications. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4477–4482. IEEE, 2016.

[7] Ruffin White, Dr Henrik I Christensen, and Dr Morgan Quigley. Sros: Securing ros over the wire, in the graph, and through the kernel. *arXiv preprint arXiv:1611.07060*, 2016.

[8] Gianluca Caiazza, Ruffin White, and Agostino Cortesi. *Enhancing Security in ROS*, pages 3–15. Springer Singapore, Singapore, 2019.

[9] Victor Mayoral-Vilches, Ruffin White, Gianluca Caiazza, and Mikael Arguedas. Sros2: Usable cyber security tools for ros 2. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 11253–11259, 2022.

[10] ROS2 core team. 2023-09 ros 2 rmw alternate. https://discourse.ros.org/t/ros-2-alternative-middleware-report/33771.

[11] Giacomo Zanatta, Pietro Ferrara, Gianluca Caiazza, Teodors Lisovenko, Luca Negrini, and Ruffin White. Sound static analysis for microservices: Utopia? a preliminary experience with lisa. In *Proceedings of the 26th ACM International Workshop on Formal Techniques for Java-like Programs*. ACM, 2024.

[12] Luca Negrini, Pietro Ferrara, Vincenzo Arceri, and Agostino Cortesi. *LiSA: A Generic Framework for Multilanguage Static Analysis*, pages 19–42. Springer Nature Singapore, Singapore, 2023.

[13] Pietro Ferrara, Luca Negrini, Vincenzo Arceri, and Agostino Cortesi. Static analysis for dummies: experiencing lisa. In *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, SOAP 2021, page 1–6, New York, NY, USA, 2021. Association for Computing Machinery.

[14] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[15] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *6th Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282. ACM Press, 1979.

[16] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. A suite of abstract domains for static analysis of string values. *Softw. Pract. Exp.*, 45(2):245–287, 2015.

[17] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[18] Pietro Ferrara and Luca Negrini. SARL: OO framework specification for static analysis. In *12th International Conference on Verified Software: Theories, Tools, and Experiments - VSTTE 2020,*, volume 12549 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2020.

[19] Minh Ngoc Ngo and Hee Beng Kuan Tan. Applying static analysis for automated extraction of database interactions in web applications. *Information and Software Technology*, 50(3):160–175, 2008.

[20] B. J. Berger, R. Nguempnang, K. Sohr, and R. Koschke. Static extraction of enforced authorization policies seeauthz. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 187–197, Los Alamitos, CA, USA, oct 2020. IEEE Computer Society.

[21] André Santos, Alcino Cunha, and Nuno Macedo. The high-assurance ros framework. In *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*, pages 37–40, 2021.

[22] André Santos, Alcino Cunha, Nuno Macedo, and Cláudio Lourenço. A framework for quality assessment of ros repositories. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4491–4496, 2016.

[23] Yanan Liu, Yong Guan, Xiaojuan Li, Rui Wang, and Jie Zhang. Formal analysis and verification of dds in ros2. In *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 1–5, 2018.

[24] Ivan Perez, Anastasia Mavridou, Tom Pressburger, Alexander Will, and Patrick J Martin. Monitoring ros2: from requirements to autonomous robots. *arXiv preprint arXiv:2209.14030*, 2022.

[25] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu. Rosrv: Runtime verification for robots. In *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings 5*, pages 247–254. Springer, 2014.

[26] Angelo Ferrando, Rafael C Cardoso, Michael Fisher, Davide Ancona, Luca Franceschini, and Viviana Mascardi. Rosmonitoring: a runtime verification framework for ros. In *Towards Autonomous Robotic Systems: 21st Annual Conference, TAROS 2020, Nottingham, UK, September 16, 2020, Proceedings 21*, pages 387–399. Springer, 2020.

[27] Sorin Adam, Morten Larsen, Kjeld Jensen, and Ulrik Pagh Schultz. Towards rule-based dynamic safety monitoring for mobile robots. In *Simulation, Modeling, and Programming for Autonomous Robots: 4th International Conference, SIMPAR 2014, Bergamo, Italy, October 20-23, 2014. Proceedings 4*, pages 207–218. Springer, 2014.

[28] R Rohith, Minal Moharir, G Shobha, et al. Scapy-a powerful interactive packet manipulation program. In *2018 international conference on networking, embedded and wireless systems (ICNEWS)*, pages 1–5. IEEE, 2018.

[29] Christophe Bédard, Pierre-Yves Lajoie, Giovanni Beltrame, and Michel Dagenais. Message flow analysis with complex causal links for distributed ros 2 systems. *Robotics and Autonomous Systems*, 161:104361, 2023.

[30] Gelei Deng, Guowen Xu, Yuan Zhou, Tianwei Zhang, and Yang Liu. On the (in)security of secure ros2. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 739–753, New York, NY, USA, 2022. Association for Computing Machinery.

[31] OSRF. https://github.com/osrf/nodl_to_policy, Accessed on 2024-02-28.

[32] ROS2 Design. Design node interface definition language (idl). https://github.com/ros2/design/pull/266, Accessed on 2024-02-28.