International Journal on Software Tools for Technology Transfer (2025) 26:797–821
https://doi.org/10.1007/s10009-024-00777-8

GENERAL

Special Issue: CSV 2024

Inference of access policies through static analysis

Giacomo Zanatta[1] · Gianluca Caiazza[1] · Pietro Ferrara[1] · Luca Negrini[1]

Accepted: 12 December 2024 / Published online: 10 January 2025

Abstract
Robot Operating System 2 (ROS 2) is the de-facto standard framework for developing distributed robotic applications. However, ensuring the correctness and security of these applications remains a significant challenge. This paper presents a novel approach to statically analyze ROS 2 applications using abstract interpretation. By extracting the architecture graph of the application, our method derives minimal access control policies that can be used to leverage security. We implemented our approach using the Library for Static Analysis (LiSA), providing a toolset that facilitates the development of sound static analyzers for ROS 2. The results demonstrate the effectiveness of our approach in enhancing the security of ROS 2 applications.

Keywords Static Analysis · Abstract Interpretation · Access Policies · Robotics · Cybersecurity

1 Introduction

Robotic applications, Industrial Control Systems (ICS), and, more broadly, Cyber-Physical Systems (CPS) are becoming increasingly prevalent, seamlessly integrating into various aspects of our daily lives, including safety-critical tasks. From autonomous vehicles to teleoperated medical robots, these systems perform essential functions where human lives and environmental protection are at stake. Therefore, it is crucial to thoroughly assess how system malfunctions or misbehavior could compromise or jeopardize safety [1].

Historically, robotics development has prioritized functionality and task completion over security, with no design effort spent defending against malicious actors [2]. This approach was sufficient when these systems operated in controlled, greenfield environments (e.g., laboratories) where security risks were minimal, interactions were closely monitored, and connectivity to third parties was either limited or nonexistent. However, as these devices transition into real-world products and are deployed in uncontrolled, brownfield environments, safety concerns have grown significantly [3].

Robot Operating System (ROS) [4] and its successor, ROS 2 [5], have become the *de facto* standard frameworks for robotic development. In recent years, the importance of security within the ROS community has grown. A 2022 community survey revealed that 73% of participants recognized the need for increased investment in protecting their robots from cyber threats [6]. Despite this awareness and openness to investment, only 26% of participants reported having actually invested in such security measures. This disparity highlights a critical challenge: while the market acknowledges the necessity of robust security, the complexity of determining the most effective path forward often leads to hesitation. Without a clear sense of which solutions will provide the greatest return on investment (ROI), organizations may delay or limit their security spending, leaving systems vulnerable even as the need for protection becomes increasingly urgent.

We consider ROS 2 to be an excellent candidate for our research not only due to its widespread market adoption but also because it exemplifies the principles seen in other modern robotic and CPS protocols. For instance, protocols like Eclipse Zenoh and various Internet of Things (IoT) solutions like Apple HomeKit, Google Home, and the Matter Alliance rely on complex, distributed architectures with loosely coupled graph representations. This architectural abstraction simplifies development, enhances code portability across different versions and technology solutions, and supports our goal of achieving hardware-agnostic security.

✉ G. Zanatta
giacomo.zanatta@unive.it

G. Caiazza
gianluca.caiazza@unive.it

P. Ferrara
pietro.ferrara@unive.it

L. Negrini
luca.negrini@unive.it

[1] Ca' Foscari University, Venice, Italy

Currently, ROS 2 builds upon the Data Distribution Service (DDS) Security (DDS-Security) specifications, implemented within the ROS Client Library (RCL) to provide security features. With Secure ROS 2 (SROS2) [7], developers can access utilities that help set up security configurations that the RCL can use. These tools ease security deployment by managing tasks such as handling Certificate Authorities (CAs), identity artifacts for nodes, and governance files while also translating ROS 2 security terms into low-level DDS permissions.

As of this writing, the ROS community is actively working on supporting alternative middleware solutions to complement Secure DDS.[1] In this effort, security features such as Encryption, Authentication, and Access Control have been identified as "must-have" requirements (as defined in RFC 2119) [8].

In ROS 2, constructing security policies is a complex and error-prone task due to the challenges of manually capturing all possible interactions within the system. Currently, developers rely on exhaustive testing, which involves creating a series of test cases to simulate a range of runtime behaviors. Each test captures a snapshot of active nodes, topics, services, and actions observed during that specific execution. Developers then merge these snapshots, combining data across test cases to approximate the complete set of interactions for a policy. However, this merging process is inherently limited to the conditions covered by the tests, often missing interactions that could occur in different runtime scenarios.

A correct policy, in this context, is one that adheres to the Principle of Least Privilege, allowing each node to interact only with the minimum set of resources necessary for its function [44]. Manual testing-based approaches risk over- or under-provisioning permissions, potentially creating either insecure or functionally restrictive policies. Our tool, LiSA4ROS2, addresses this by applying static analysis to exhaustively approximate all possible interactions, generating security policies that meet this correctness criterion without relying solely on runtime testing.

As policies evolve over time—due to system updates, added functionalities, or new security requirements—maintaining correctness becomes increasingly difficult. When multiple developers contribute to the process, the risk of inconsistencies and oversights rises, leading to policies that may either overgrant or undergrant permissions. These issues are exacerbated by the fact that each developer may have different interpretations of the system's needs, further complicating the effort to maintain a unified and minimal policy set.

We argue that relying solely on the skills and intuition of developers to establish security policies is not a sustainable or effective approach. More usable, automated solutions are needed to reduce the likelihood of human error and ensure that policies remain correct, consistent, and aligned with security best practices, even as they evolve over time.

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

Contributions To address these challenges, we introduce LiSA4ROS2 [9, 10], a tool for the automatic extraction of security policies from source code via abstract interpretation. This tool performs a sound static analysis over ROS 2 Python applications, computing an overapproximation of the architectural graph (named ROS Computational Graph) and generating a correct and minimal configuration for ROS 2 security policies. In this paper, which extends our previous work [9], we provide a formalization of the tool.

LiSA4ROS2 is implemented as an extension of the Library for Static Analysis (LiSA) [11, 12]. Starting from an ROS 2 Python application, LiSA4ROS2 extracts the program's ROS Computational Graph and generates an XML-based access policy specification. We evaluated LiSA4ROS2 using both official tutorial code and real-world ROS 2 Python applications from GitHub, covering all ROS 2 communication patterns.

LiSA4ROS2 is fully automatic, and its integration with SROS2 within the DevSecOps model enhances both usability and security. Moreover, it can be generalized for use in other robotics and CPS ecosystems where graph-based architectures are applied across multiple programming languages. The tool, along with Docker images and results, is available in the following GitHub repositories:

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

Outline Section 2 gives some background about the Robot Operating System 2 (ROS 2) framework, proposing a running example that will accompany us throughout the reading. Section 3 sets up the formalization of the paper. Section 4 presents our static analysis tool, LiSA4ROS2. In Sect. 5 we evaluate the tool on a set of selected examples. Related and future work are presented, respectively, in Sects. 6 and 7. Section 8 concludes. Two appendices discuss the Data Distribution Service (DDS) permissions, while in the last one we present an algorithm for extracting the policies.

2 Robot Operating System 2 (ROS 2)

The Robot Operating System 2 (ROS 2) framework [5] is a powerful suite of libraries and tools for building distributed robotic applications. Its predecessor, ROS 1 [4], quickly established itself as a popular framework for robotic research and development. However, despite its success, ROS 1 had

several limitations, particularly in terms of real-time performance, scalability, security, and support for multirobot systems. These challenges became more evident as the robotics field evolved and more complex, safety-critical applications emerged.

To address these shortcomings, ROS 2 was introduced in 2015. Built from the ground up, ROS 2 was designed to overcome the limitations of its predecessor by incorporating modern middlewares (e.g., DDS) to provide improved support for real-time control, enhanced security, greater flexibility in network configurations, and better multiplatform compatibility.

At the core of an ROS 2 application is a collection of processes, known as nodes, that collaborate by exchanging messages to achieve a shared objective. The behavior of these nodes is defined using a client library, a lightweight, high-level API that simplifies the creation of ROS 2 applications. These libraries abstract away the complexities of the underlying framework, enabling developers to concentrate on application logic while the internal communication mechanisms are seamlessly managed.

At the time of writing, ROS 2 officially supports two primary client libraries: rclpy[2] for Python and rclcpp[3] for C++. These libraries allow developers to build and define nodes in either Python or C++, offering flexibility based on the project's requirements.

In the following, we will provide informal definitions of key ROS 2 concepts to further clarify the framework's fundamental components.

Node In ROS 2, a node is a fundamental computational unit, essentially a distinct process running on a machine. Each node is typically responsible for a specific function within the robot's control system, such as processing sensor data, controlling actuators, performing path planning, or managing communication. Nodes are the building blocks of modular robotic systems, allowing developers to break down complex tasks into smaller, more manageable components while enabling the reuse and sharing of preexisting modules.

For example, a node might interface with sensors to collect real-time data from the external environment, such as temperature, distance, or visual input—and then process this information to make informed decisions. Similarly, nodes can issue commands to actuators, enabling physical actions like moving the robot's arms or wheels, executing navigation tasks, or dynamically adjusting behavior based on changing environmental conditions.

Each node is uniquely identified by a name and can be organized within a namespace, creating a hierarchical structure that helps manage complexity in larger systems, much like the tree structure of a file system. The use of namespaces serves several key purposes: (i) it prevents naming conflicts by allowing multiple nodes with the same base name to coexist in different contexts; (ii) supports modular development by grouping related nodes together, and enhances clarity regarding how different system components interact; (iii) enables seamless integration of additional nodes or subsystems, by allowing for easy addition, removal, or replacement of nodes without disrupting or adapting the overall architecture by enforcing separation of concerns between the robot's functionality.

Communication mechanisms Nodes communicate with each other by exchanging messages. Depending on the use case, communication can occur through different mechanisms:

- **Topics.** A topic is a named channel that facilitates internode communication. Nodes can publish messages to a topic or subscribe to receive messages from it. Topics support asynchronous and decoupled communication, making them ideal for scenarios where data needs to be shared across multiple nodes.
- **Services.** For synchronous communication, nodes can utilize services. Services provide a request-response communication model between two nodes. In this model, a node acts as a server (hosting the service), while other nodes, acting as clients, can request information or perform operations by connecting to the server. Services are suited for interactions requiring immediate feedback or when direct communication between nodes is needed, such as executing critical tasks.
- **Actions.** Actions also use a client/server model but are tailored for long-running tasks that may need to be preempted. An action client sends a goal to an action server, specifying a task to be executed (e.g., reaching a specific coordinate). The server periodically sends feedback messages to inform the client about the progress of the task (e.g., "I am X meters from the destination"). Upon completion, the server sends the final result to the client (e.g., "I have reached the destination"). Clients have the ability to cancel ongoing actions if necessary.

ROS computational graph The interactions between nodes in an ROS 2 system are represented by a specific network graph known as the ROS Computational Graph. This graph models the network of nodes and their connections within the system. It provides a structural representation that helps in understanding how nodes interact with each other and in verifying the correctness of the architectural design of an ROS 2 application. An example of this type of graph is shown in Fig. 1, which will be discussed in detail in the following section. Although the term computational graph may have different interpretations in other fields, in this work, we stick to the terminology used by the framework.

[2] https://github.com/ros2/rclpy.

[3] https://github.com/ros2/rclcpp.

Fig. 1 Extracted graph of the minimal Talker–Listener example

2.1 Talker–listener ROS 2 Python example

As a running example for our paper, we introduce the Talker–Listener ROS 2 Python example. This example is part of the official ROS 2 documentation[4] and serves as a minimal program demonstrating communication between two nodes. It is used as a tutorial to explain how to set access controls in ROS 2.[5]

The code provided in Listing 1 defines a Talker class (line 6) that extends the `Node` object from the `rclpy` library, making instances of this class fully functional ROS 2 nodes. Specifically, within the constructor of the Talker class (line 7), the constructor of the superclass, i.e., `rclpy`'s `Node` (line 8), is called. This constructor creates the node, registers it with the ROS 2 system, and enables it to send and receive messages. The string parameter passed to the superclass constructor specifies the node's name (`talker`). Line 10 defines a publisher using the `create_publisher` statement. This statement declares a topic named `chatter` (the second parameter) if it does not already exist in the ROS Computational Graph and instantiates a publisher for sending messages of type `String` over that topic (the first parameter). The third parameter specifies the Quality of Service settings for the publisher, which is beyond the scope of this explanation. Continuing with the constructor implementation, line 12 defines a timer that triggers the execution of a function (the second parameter, `timer_callback`) every `timer_period` seconds. The callback creates a new `String` message (lines 15–16) and publishes it using the previously created publisher (line 19). In the main function, we (i) initialize the ROS 2 environment (line 23), (ii) instantiate the node (line 24), and (iii) let the ROS 2 framework manage the node (line 25).

Listing 2 illustrates a similar approach. Here, we create a node named `listener` (line 8). Instead of a publisher, this node uses the `create_subscription` statement (line 9) to create a subscriber for the same `chatter` topic as the `talker` node. This setup enables communication between the two nodes. The third parameter of the `create_subscription` call specifies the callback function, which is triggered for every message received from the topic. The ROS Computational Graph for this example is depicted in Fig. 1, where ovals represent nodes and rectangles represent topics. An arc from a node to a topic indicates that the node publishes to that topic, while an arc from a topic to a node indicates that the node subscribes to that topic.

```
1import rclpy
2from rclpy.executors import ExternalShutdownException
3from rclpy.node import Node
4from std_msgs.msg import String
5
6class Talker(Node):
7  def __init__(self):
8    super().__init__('talker')
9    self.i = 0
10   self.pub = self.create_publisher(String, 'chatter', 10)
11   timer_period = 1.0
12   self.tmr = self.create_timer(timer_period,
           self.timer_callback)
13
14 def timer_callback(self):
15   msg = String()
16   msg.data = 'Hello World: {0}'.format(self.i)
17   self.i += 1
18   self.get_logger().info('Publishing:"{0}"'.format(msg.data))
19   self.pub.publish(msg)
20
21def main(args=None):
22 try:
23   with rclpy.init(args=args):
24     node = Talker()
25     rclpy.spin(node)
26 except (KeyboardInterrupt, ExternalShutdownException):
27   pass
28if __name__ == '__main__':
29 main()
```

Listing 1 Python's ROS 2 Talker example

```
1import rclpy
2from rclpy.executors import ExternalShutdownException
3from rclpy.node import Node
4from std_msgs.msg import String
5
6class Listener(Node):
7  def __init__(self):
8    super().__init__('listener')
9    self.sub = self.create_subscription(String, 'chatter',
           self.chatter_callback, 10)
10 def chatter_callback(self, msg):
11   self.get_logger().info('I heard: [%s]' % msg.data)
12
13def main(args=None):
14 try:
15   with rclpy.init(args=args):
16     node = Listener()
17     rclpy.spin(node)
18 except (KeyboardInterrupt, ExternalShutdownException):
19   pass
20if __name__ == '__main__':
21 main()
```

Listing 2 Python's ROS 2 Listener example

2.2 ROS 2 layered architecture

ROS 2 is designed as a modular, layered framework that abstracts internal complexities from the user. This layered approach divides the system into distinct functional segments, each with its own responsibilities, allowing developers to implement applications agnostic to the underlying communication middleware and making the framework more adaptable and versatile.

[4] Available at https://github.com/ros2/demos/tree/rolling/demo_nodes_py/demo_nodes_py/topics.

[5] https://docs.ros.org/en/jazzy/Tutorials/Advanced/Security/Access-Controls.html.

The importance of this architecture is well-illustrated in the paper by Macenski et al. [5], but to provide a more intuitive understanding, consider the Talker example discussed in the previous section. When creating a publisher (line 10), the `create_publisher` function does not directly handle the registration of the publisher in the network. Instead, it delegates this task to an internal layer known as the `rcl` (ROS 2 Client Library) layer.

The `rcl` layer serves as an intermediary between the high-level client libraries, such as `rclpy` (for Python) and `rclcpp` (for C++), and the lower-level communication layer. These client libraries essentially act as wrappers for the `rcl` functions, allowing developers to write code without needing to worry about the intricacies of network communication or middleware selection. The `rcl` library, in turn, hands off communication tasks to an even lower-level layer called the `rmw` (ROS 2 Middleware) layer.

The `rmw` layer plays a crucial role in decoupling ROS 2 from specific communication middleware implementations. It defines a set of standardized APIs for communication operations, such as creating publishers, subscribers, services, and clients. Different middleware implementations can be plugged in underneath the `rmw` layer, enabling ROS 2 to operate independently of the specific middleware in use. This flexibility allows ROS 2 applications to remain middleware-agnostic, providing developers with the freedom to select the middleware that best meets the performance, latency, or feature requirements of their particular application.

Several `rmw` implementations are available, each corresponding to different DDS (Data Distribution Service) vendors. For example, ROS 2 supports rmw implementations like Fast-DDS (`rmw_fastrtps_cpp`) and Cyclone DDS (`rmw_cyclonedds_cpp`). By abstracting away middleware-specific details, ROS 2 enables developers to change middleware implementations without modifying their application code. This can be done simply by adjusting configuration settings, typically through environment variables, at runtime. As a result, the same ROS 2 application can run on different middleware backends, offering flexibility for a wide range of use cases.

Currently, Fast-DDS is the default middleware in ROS 2 and is also the most widely used, owing to its balance of performance, feature set, and ease of configuration. However, the modular architecture of ROS 2 means that developers can easily switch to other middleware options as needed, making ROS 2 a highly adaptable framework for robotics and distributed systems.

2.3 Securing an ROS 2 application

Securing an ROS 2 application requires the explicit definition of ROS 2 security policies that are subsequently mapped to the Data Distribution Service (DDS) specifications. To streamline this process and provide a standardized approach to security policy management, agnostic with respect to the underlying communication middleware, the framework provides a powerful command-line tool known as Secure ROS 2 (SROS2) [7]. SROS2 acts as a versatile toolset for implementing various security configurations. It enables developers to generate security artifacts such as keys, certificates, and permission files, which are necessary for enforcing Access Control and Encryption. Additionally, SROS2 automates many of the complex steps involved in securing ROS 2 nodes (e.g., Certificate Authority (CA) management, Keystore), making it easier to integrate security features into an application without requiring deep expertise in cybersecurity deployment practices.

Developers are still responsible for configuring access control, which involves manually defining specific permissions for each node, detailing which topics, services, or actions they are allowed to access or perform. This fine-grained access control ensures that only authorized nodes can communicate or interact with particular system components, thereby reducing the attack surface of the application. For Encryption, which secures data in transit by rendering it unreadable to unauthorized parties, DDS enforces this protection automatically once the ROS' security flag is enabled and the appropriate security artifacts (generated by SROS2) are correctly configured and provisioned. This ensures that sensitive information remains protected from eavesdropping or tampering, thereby enhancing the overall security of the ROS 2 application.

In the following, we will explain in detail how access policies in ROS 2 are defined and we will see how to applying these security measures in a practical running example.

2.3.1 Access policies

In SROS2, access policies are implemented through XML files. The structure begins with a set of enclaves, each identified by a unique path. The rule of thumb is that we group all the nodes with similar functionalities or security needs in an enclave, establishing logical boundaries for centralized security management.

Each enclave contains a set of rules known as a profile, which is associated with individual nodes identified by their names and namespaces. These profiles outline permissions for all the different types of entities in ROS 2, such as topics, services, and actions. Permissions are managed through specific XML tags for each entity type, with each tag marked with `ALLOW` or `DENY`.

`ALLOW` explicitly grants access to the specified entities or actions. However, to provide more granular control, `DENY` is used to restrict or override the permissions granted by `ALLOW`. Essentially, `DENY` refines the scope of `ALLOW` by blocking

Listing 3 Subscriber Permission example

```
<topics subscribe="ALLOW">
    <topic>/foo/*</topic>
    <topic>/bat</topic>
</topics>
<topics subscribe="DENY">
    <topic>/foo/bar</topic>
</topics>
```

specific operations or resources that would otherwise be permitted. For example, while `ALLOW` might permit access to all topics in a namespace, a subsequent `DENY` can be used to block access to particular topics within that broader permission set. The rest of the behavior for resource access adheres to the 'Deny by default' principle, where only explicitly authorized actions are permitted, and all others are implicitly blocked.

In particular, we can model the following tags:

- **Topics.** XML tags for topics specify permissions for publishing and subscribing. For instance, `<topics publish="ALLOW">` permits a node to publish to the listed topics that follow the tag, while `<topics subscribe="ALLOW">` permits subscribing to those topics. Conversely, `<topics publish="DENY">` and `<topics subscribe="DENY">` block publishing or subscribing to the specified topics.
- **Services.** `<services reply="ALLOW">` and `<services request="ALLOW">` define whether a node can act as a service server or client, respectively. Tags marked with `DENY` restrict these roles by blocking the node from acting as a service server or client for the specified services.
- **Actions.** `<actions execute="ALLOW">` and `<actions call="ALLOW">` allow a node to be an action server or client, respectively. Corresponding tags marked with `DENY` block these roles.

Additionally, SROS2 policy files permit the usage of basic fnmatch-style patterns:

- `*` for matching everything;
- `?` for matching any single character;
- `[sequence]` for matching any character in the sequence;
- `[!sequence]` for matching any character not in the sequence.

The snippet in Listing 3 illustrates a set of permissions that allows the node to subscribe to all topics within the namespace rooted at */foo* (line 2) and to the topic */bat*, located at the root level (line 3). However, it explicitly denies subscription access to the */foo/bar* topic (line 6) within the *foo* namespace, thereby refining the granted permissions with more granular control.

2.3.2 Minimal access control policies

An access control policy is considered minimal if it grants the least number of permissions necessary for nodes to perform their required functions, adhering to the Principle of Least Privilege (PoLP). This ensures that no extraneous access rights are provided beyond what is strictly needed.

```
rosdev@testbed:~/ros2_ws$ ros2 security create_keystore
    keystore
rosdev@testbed:~/ros2_ws$ ros2 security create_enclave
    keystore /talker_listener/talker
rosdev@testbed:~/ros2_ws$ ros2 security create_enclave
    keystore /talker_listener/listener
export ROS_SECURITY_KEYSTORE=~/ros2_ws/keystore
export ROS_SECURITY_ENABLE=true
export ROS_SECURITY_STRATEGY=Enforce
```

Listing 4 ROS 2 Security setup

2.4 Running example

In the following section, we will dive into the practical implementation of access policies and demonstrate the application of security measures using the Talker–Listener example introduced in Sect. 2.1. We will illustrate how permissions are configured for each node and explain the process of setting up encryption to safeguard data in transit, thereby ensuring a secure communication channel between the nodes.

The first step in securing these nodes is to create a keystore and configure the ROS 2 system to use it, as shown in Listing 4.

In detail, we first create a keystore in the current directory to store the public and private keys required for securing our ROS 2 application. The command `create_keystore keystore` initializes the keystore. Next, two separate enclaves are created for the Talker and Listener nodes using `create_enclave keystore /talker_listener/talker` and `create_enclave keystore /talker_listener/listener`, respectively. As previously mentioned, an enclave groups nodes with common security policies, and in this case, each node is assigned to its own enclave, allowing for more tailored security measures.

To enable and enforce security, we export three environment variables:

- `ROS_SECURITY_KEYSTORE` specifies the location of the keystore.
- `ROS_SECURITY_ENABLE=true` activates security mode, ensuring that ROS 2 applies security policies when running the nodes.
- `ROS_SECURITY_STRATEGY=Enforce` strictly enforces these policies. If a node does not conform to the specified rules, it will fail to launch, ensuring only compliant nodes communicate within the system.

These steps establish a secure environment, enabling access control and encryption to protect the data exchange.

During this process, SROS2 automatically generates two dummy DDS permission XML files (see Appendix A), one for the Talker node and one for the Listener node. However, these files allow by default the nodes to publish or subscribe

```
<policy version="0.2.0">
<enclaves>
<enclave path="">
  <profiles>
    <profile node="listener" ns="">
      <topics subscribe="ALLOW">
        <topic>/chatter</topic>
      </topics>
      </profile>
    </profiles>
  </enclave>
</enclaves>
</policy>
```

Listing 5 Policy generated by the SROS2 security tool for the Listener node

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<policy version="0.2.0">
 <enclaves>
   <enclave path="">
        <profiles>
          <profile ns="" node="talker">
            <topics publish="ALLOW">
                 <topic>/chatter</topic>
               </topics>
          </profile>
        </profiles>
   </enclave>
 </enclaves>
</policy>
```

Listing 6 Policy generated by SROS2 for the Talker node

to any topic. It is the developer's responsibility to refine these policies by editing the rules and minimizing permissions, ensuring that nodes have access only to the necessary resources.

While this is relatively straightforward for a simple project, it becomes laborious when dealing with large-scale systems with numerous nodes and complex communication patterns.

Instead of manually writing these policies, SROS2 offers a tool to automatically generate them by inspecting the ROS 2 network. This process requires temporarily disabling security by setting to false `ROS2_SECURITY_ENABLE` to allow node discovery. The policy generated reflects the actual activity and interactions of the system; it captures all topics, services, and actions that the nodes have utilized during sampling. However, this approach results in a single, generalized policy that encompasses the entire ROS graph, which does not align with the PoLP. To achieve proper segmentation and ensure that each node only accesses the communication channels relevant to its role, we must individually run each node. By doing so, SROS2 can generate a tailored policy, capturing nodes' specific interactions and access requirements.

Below, we showcase how to do it for the Listener node, for which we need to generate a security policy. At first, we run SROS2 on the live Listener node as follows:

```
rosdev@testbed:~/ros2_ws$ ros2 security generate_policy
    policy.xml
```

As a result, this command creates an XML policy file as shown in Listing 5.[6]

Once the policy has been created, we can generate the corresponding security artifacts by invoking the following command:

```
rosdev@testbed:~/ros2_ws$ ros2 security generate_artifacts -k
    keystore -p policy.xml
```

[6] To streamline the discussion, we omit internal services and topics used by the ROS 2 node, which are not explicitly defined in the application code. The full policy, including all internal communications, is provided in Appendix B.

```
rosdev@testbed:~/ros2_ws$ ros2 run talker_listener listener
    --ros-args --enclave /talker_listener/listener
[INFO] [1724175637.573096471] [listener]: I heard: "Hello
    World: 37"
[INFO] [1724175638.053419896] [listener]: I heard: "Hello
    World: 38"
[INFO] [1724175638.553080667] [listener]: I heard: "Hello
    World: 39"
[INFO] [1724175639.052854812] [listener]: I heard: "Hello
    World: 40"
[INFO] [1724175639.552343915] [listener]: I heard: "Hello
    World: 41"
[INFO] [1724175640.052002816] [listener]: I heard: "Hello
    World: 42"
[INFO] [1724175640.553140573] [listener]: I heard: "Hello
    World: 43"
```

Listing 7 Example of execution of Listener node with security enabled

```
def chatter_callback(self, msg):
 self.get_logger().info('I heard: [%s]' % msg.data)
 if msg.data == "Hello World: 100":
   self.publisher = self.create_publisher(String, "chatter2",
       10)
   msg = String()
   msg.data = "Hello World from listener node"
   self.publisher.publish(msg)
```

Listing 8 A more complex chatter_callback function

After provisioning the Listener node and generating its associated policy, we proceed by applying the same steps to the Talker. The policy generated for the Talker node is shown in Listing 6.

With both nodes' policies correctly established and integrated, and security fully enabled, we can run the Talker and Listener nodes in secure mode. Upon execution, the Listener successfully receives messages from the `chatter` topic, confirming that secure communication between the nodes is functioning as intended and showed in Listing 7.

We now introduce an additional layer of complexity to this example. Specifically, we modify the Listener class to publish a message on the `chatter2` topic if it receives the message `"Hello world: 100"`. This behavior is implemented by updating the `chatter_callback` function, as shown in Listing 8.

However, since the Listener node lacks the necessary permissions to publish on the `chatter2` topic, the ROS 2 process managing the Listener will fail. This failure occurs be-

```
[INFO] [1724175918.929412805] [listener]: I heard: "Hello
    World: 96"
[INFO] [1724175919.429809460] [listener]: I heard: "Hello
    World: 97"
[INFO] [1724175919.929810487] [listener]: I heard: "Hello
    World: 98"
[INFO] [1724175920.428716769] [listener]: I heard: "Hello
    World: 99"
[INFO] [1724175920.928859535] [listener]: I heard: "Hello
    World: 100"
2024-08-20 17:45:20.929 [SECURITY Error] rt/chatter2 topic
    not found in allow rule.
    (./src/cpp/security/accesscontrol/Permissions.cpp:1191)
    -> Function check_create_datawriter
2024-08-20 17:45:20.929 [SECURITY Error] Error checking
    creation of local writer (rt/chatter2 topic not found in
    allow rule.
    (./src/cpp/security/accesscontrol/Permissions.cpp:1191))
-> Function get_datawriter_sec_attributes
2024-08-20 17:45:20.929 [DATA_WRITER Error] Problem creating
    associated Writer -> Function enable
```

Listing 9 Permission failure with security enforced

cause the system attempts an unauthorized publishing operation, which is blocked by the enforced security policies, as illustrated in Listing 9.

Since the publish declaration in the Listener node is created only upon receiving a specific message, generating the policy with SROS2 before this message is received will exclude the publishing capability from the policy. This results in an incomplete and broken policy. Even if we were to regenerate the policies from scratch, the issue might persist. This is because the regenerated policy would still only capture the system's state at the moment of its generation, potentially missing dynamically created components or interactions that occur after the policy has been generated. This highlights a limitation of generating permissions through dynamic inspection, which can fail when access to a channel is nondeterministic. To overcome this issue, static analysis can be employed. Our tool, LiSA4ROS2, is capable of extracting these communication links ahead of time. Suppose that a developer has built the LiSA4ROS2 project by following the README in the official repository; the tool can generate the correct SROS2 policies for both the Talker and Listener nodes by running the following command:

```
./gradlew run --args="talker.py listener.py /talker_listener
    output"
```

The generated policy for the Listener node is stored in `output/policies/listener/policy.xml`, as shown in Listing. 10.

Notably, the `chatter2` topic is now included in the policy, addressing the issue of missing permissions. This ensures that the Listener node can correctly publish to `chatter2` when required. The updated ROS Computational Graph, which includes the `chatter2` topic, is illustrated in Fig. 2.

With the newly generated policy in place, which now includes the necessary permissions and the artifacts generated by the SROS2 toolchain, the Listener node is equipped with the proper authorization. Consequently, it can successfully

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<policy version="0.2.0">
<enclaves>
  <enclave path="">
    <profiles>
      <profile ns="/" node="listener">
        <topics publish="ALLOW">
          <topic>/chatter2</topic>
        </topics>
        <topics subscribe="ALLOW">
          <topic>/chatter</topic>
        </topics>
      </profile>
    </profiles>
  </enclave>
</enclaves>
</policy>
```

Listing 10 Policy generated by LiSA4ROS2 for the Listener node

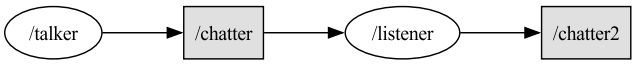

Fig. 2 Extracted graph of the minimal example with the chatter2 topic

```
[INFO] [1724178848.095704191] [listener]: I heard:
    "Hello World: 95"
[INFO] [1724178848.596561313] [listener]: I heard:
    "Hello World: 96"
[INFO] [1724178849.095332576] [listener]: I heard:
    "Hello World: 97"
[INFO] [1724178849.596307954] [listener]: I heard:
    "Hello World: 98"
[INFO] [1724178850.096445124] [listener]: I heard:
    "Hello World: 99"
[INFO] [1724178850.596089405] [listener]: I heard:
    "Hello World: 100"
[INFO] [1724178850.598107446] [listener]: Publishing
    on chatter2: "Hello World from listener node"
[INFO] [1724178851.095993233] [listener]: I heard:
    "Hello World: 101"
[INFO] [1724178851.596415870] [listener]: I heard:
    "Hello World: 102"
[INFO] [1724178852.095364262] [listener]: I heard:
    "Hello World: 103"
[INFO] [1724178852.596199131] [listener]: I heard:
    "Hello World: 104"
```

Listing 11 Correct execution of the modified Listener node

publish messages on the `chatter2` topic without encountering any security-related errors, as shown in Listing 11.

3 Formalization

In this section, we formalize our static analysis, how ROS Computational Graphs are extracted from the static analysis results, and how access policies are extracted from ROS Computational Graphs.

3.1 Notation and background

In the rest of this section, we will denote sets with names starting with an upper-case character and elements in sets with lower-case characters. Given two sets A and B, we will

```
exp   ::=  sexp |
           hexp |
           x. < methodname > |
           x. < methodname > (par_1, ..., par_n)
sexp  ::=  " < string > " | x + y | ...
hexp  ::=  new C | x.f
st    ::=  x = exp | x.f = exp | exp
```

Fig. 3 Minimal language

denote the Cartesian product of these two sets by $\mathsf{A} \times \mathsf{B}$, and functions with domain A and codomain B by $\mathsf{A} \to \mathsf{B}$. Given a set A, the powerset of A is denoted by $\wp(\mathsf{A})$.

We denote by $\langle \mathsf{A}, \leq, \bot, \top, \sqcup, \sqcap \rangle$ a lattice of elements in A with $\leq$ as partial ordering operator, $\bot$ as bottom element, $\top$ as top element, $\sqcup$ as the least upper bound operator, and $\sqcap$ as the greatest lower bound operator.

Our approach relies on the abstract interpretation framework [13, 14]. We will denote concrete sets and elements by C and c, respectively, and abstract sets and elements by $\overline{\mathsf{A}}$ and $\overline{\mathsf{a}}$, respectively. In addition, $\gamma_{\overline{\mathsf{A}}}$ will denote the concretization function of the abstract domain $\overline{\mathsf{A}}$. An abstract domain is a sound approximation of a concrete one if they form a Galois connection; formally, $\langle \mathsf{C}, \subseteq \rangle \underset{\alpha_{\overline{\mathsf{A}}}}{\overset{\gamma_{\overline{\mathsf{A}}}}{\leftrightarrows}} \langle \overline{\mathsf{A}}, \sqsubseteq \rangle$.

The runtime semantics of a programming language is formalized as a concrete semantics on the concrete domain. In particular, given a statement st and a concrete state $\mathsf{c} \in \mathsf{C}$, the concrete semantics formalizes the results of the execution producing a concrete exit state $\mathsf{c}' \in \mathsf{C}$. We represent this semantics by $\mathbb{S}[\![\mathsf{st}, \mathsf{c}]\!] = \mathsf{c}'$. Then, the abstract semantics overapproximates the concrete one. In particular, we denote the abstract semantics by $\overline{\mathbb{S}}[\![\mathsf{st}, \overline{\mathsf{c}}]\!] = \overline{\mathsf{c}}'$.

3.2 Syntax

We focus our formalization on a minimal language that highlights all the main features of ROS 2 Python code. Figure 3 contains (a part of) the syntax of this language, and in particular:

- an expression exp might be a string expression sexp, a heap expression hexp, a pointer to a method, or a method call;
- a string expression sexp might be a constant string, the concatenation of two strings, or other expressions returning strings;[7]
- a heap expression hexp might allocate an object, or access a field; and
- a statement st might be the assignment of an expression to either a local variable or a field, or just an expression.

Statements represents the set containing all the statements of our language.

[7] We left this definition open since the variety of operations on strings is extremely wide.

3.3 Concrete domain and semantics

The definition of the concrete domain is rather standard. In particular, a state of the computation in Σ is represented as a pair of an environment in Env (relating variables in Var to values in Val) and a heap in Heap (relating references in Ref to fields in Field to values in Val). A value might be a reference in Ref, or a string in Str. Definition 1 formalizes concrete states.

Definition 1 (Concrete states)

$$\begin{aligned} \mathsf{Val} &= \mathsf{Str} \cup \mathsf{Ref} \\ \mathsf{Env} &= \mathsf{Var} \to \mathsf{Val} \\ \mathsf{Heap} &= \mathsf{Ref} \to \mathsf{Field} \to \mathsf{Val} \\ \Sigma &= \mathsf{Env} \times \mathsf{Heap} \end{aligned}$$

As usual in the abstract interpretation framework, the concrete domain is defined over a set of states. This allows the representation of user inputs, configuration parameters, and any kind of nondeterministic behavior of the program.

Definition 2 (Concrete domain)
The concrete domain is a complete lattice over a set of states using the standard set operators. Formally, $\langle \wp(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle$.

This forms a complete lattice by basic properties of set operators.

For the sake of simplicity, we assume that a standard small-step semantics of statements is provided. This semantics, given a statement as defined in Fig. 3 and a state as defined in Definition 1, returns a concrete state in Σ representing the results of the execution of the given statement on the given entry state. The concrete semantics is then defined as the pointwise application of the small step semantics over all the entry states.

Definition 3 (Concrete semantics)
Let $\to$ be the small step semantics of the language defined in Fig. 3. In particular, $\langle \sigma, \mathsf{st} \rangle \to \sigma'$ represents that this semantics, given an entry state $\sigma \in \Sigma$ and a statement st, returns an exit state $\sigma' \in \Sigma$. The collecting semantics over the concrete domain is then formalized as follows:

$$\mathbb{S}[\![\mathsf{st}, \mathsf{C}]\!] = \{\sigma' : \exists \sigma \in \mathsf{S} : \langle \mathsf{st}, \sigma \rangle \to \sigma'\}.$$

3.4 Abstract domain and semantics

Our analysis needs to have precise information about:

1. the string values computed by expressions to identify names of nodes in the robotic system, topics used to communicate, and

2. the objects allocated in the heap that represent robotic nodes in the network.

Usually, it is quite straightforward to track this information in ROS 2 programs, since nodes, channels, and listeners are statically identifiable from the code. On the one hand, we apply standard string constant propagation analyses [15]. For each variable, this analysis tracks either a bottom value (meaning that no value is possible at a given program point), a top value (meaning that any value might be possible), or a single concrete string value (meaning that in all the possible executions that variable will always have that constant value). On the other hand, we abstract heap nodes through a standard allocation-based Andersen-style heap abstraction [16]. While other analyses are possible, our implementation relies on the generic framework formalized by Ferrara [17]; we focus our formalization on the aforementioned heap abstraction.

In our analysis, we apply the so-called direct product of these domains (Definition 36.1 in [18]).

Let us assume that the following analyses are provided:

– a string constant propagation analysis

$$\overline{\mathbb{SV}} = \langle \overline{\mathsf{Str}}, \sqsubseteq_{\mathsf{Str}}, \bot_{\mathsf{Str}}, \top_{\mathsf{Str}}, \sqcup_{\mathsf{Str}}, \sqcap_{\mathsf{Str}} \rangle;$$

– a heap abstraction

$$\overline{\mathbb{H}} = \langle \overline{\mathsf{Heap}}, \sqsubseteq_{\mathsf{Heap}}, \bot_{\mathsf{Heap}}, \top_{\mathsf{Heap}}, \sqcup_{\mathsf{Heap}}, \sqcap_{\mathsf{Heap}} \rangle.$$

Given a variable, we assume that a state of these analyses returns either a top, a bottom, or a precise value. In particular, string constant propagation returns values in Str, while the heap abstraction returns an abstract heap identifier in $\overline{\mathsf{HeapId}}$. As usual in allocation-based heap abstraction, we assume that different elements in $\overline{\mathsf{HeapId}}$ approximate disjoint areas of concrete memory. Formally,

– $\forall \mathsf{x} \in \mathsf{Var}, \overline{\mathsf{s}} \in \overline{\mathsf{Str}} : \overline{\mathsf{s}}(\mathsf{x}) \in \mathsf{Str} \cup \{\bot_{\mathsf{Str}}, \top_{\mathsf{Str}}\}$;
– $\forall \mathsf{x} \in \mathsf{Var}, \overline{\mathsf{h}} \in \overline{\mathsf{Heap}} : \overline{\mathsf{t}}(\mathsf{x}) \in \overline{\mathsf{HeapId}} \cup \{\bot_{\mathsf{Heap}}, \top_{\mathsf{Heap}}\}$.

All these analyses are sound, which, following the abstract interpretation theory, means that they are equipped with a concretization and an abstract function, and they form a Galois connection with the concrete domain. Formally,

– $\langle \wp(\Sigma), \subseteq \rangle \underset{\alpha_{\overline{\mathbb{SV}}}}{\overset{\gamma_{\overline{\mathbb{SV}}}}{\leftrightarrows}} \langle \overline{\mathsf{Str}}, \sqsubseteq_{\mathsf{Str}} \rangle$;
– $\langle \wp(\Sigma), \subseteq \rangle \underset{\alpha_{\overline{\mathbb{H}}}}{\overset{\gamma_{\overline{\mathbb{H}}}}{\leftrightarrows}} \langle \overline{\mathsf{Heap}}, \sqsubseteq_{\mathsf{Heap}} \rangle$.

Definition 4 (Abstract domain)
Our abstract domain is the direct product of the string constant propagation analysis and the heap abstraction. Formally,

$$\overline{\mathbb{A}} = \overline{\mathbb{SV}} \times \overline{\mathbb{H}}.$$

Therefore, a state of our analysis is represented as a tuple $((\overline{\mathsf{s}}, \overline{\mathsf{m}}, \overline{\mathsf{t}}) \in \overline{\mathbb{A}}$ where $\overline{\mathsf{s}} \in \overline{\mathbb{SV}}$, and $\overline{\mathsf{h}} \in \overline{\mathbb{H}})$.

Lemma 1 (Soundness of the abstract domain)
The abstract domain $\overline{\mathbb{A}}$ is a sound approximation of the concrete domain. Formally,

$$\langle \wp(\Sigma), \subseteq \rangle \underset{\alpha_{\overline{\mathbb{A}}}}{\overset{\gamma_{\overline{\mathbb{A}}}}{\leftrightarrows}} \langle \overline{\mathbb{SV}} \times \overline{\mathbb{H}}, \sqsubseteq_{\mathsf{A}} \rangle,$$

where $\alpha_{\overline{\mathbb{A}}}$, $\gamma_{\overline{\mathbb{A}}}$, and $\sqsubseteq_{\mathsf{A}}$ are respectively the abstraction function, concretization function, and partial order defined as usual in direct products (Definition 36.1 in [18]).

We assume that the string constant propagation and heap analyses provide a sound abstract semantics following the abstract interpretation framework (Sect. 18.3 of [18]). In the rest of the formalization, we will denote by $\overline{\mathbb{SV}}[\![\mathsf{st}, \overline{\mathsf{s}}]\!] = (\overline{\mathsf{v}}, \overline{\mathsf{s}}')$ and $\overline{\mathbb{H}}[\![\mathsf{st}, \overline{\mathsf{h}}]\!] = (\overline{\mathsf{id}}, \overline{\mathsf{h}}')$ the application of these semantics to a given statement st and a given abstract state. The semantics might return a heap identifier or string value, respectively, if the statement returns a value.

Lemma 2 (Soundness of the abstract semantics)
We assume that $\overline{\mathbb{SV}}$ and $\overline{\mathbb{H}}$ provide the sound abstract semantics $\overline{\mathbb{SV}}[\![\_]\!]$ and $\overline{\mathbb{H}}[\![\_]\!]$, respectively. Then, the pointwise application of the two abstract semantics to the direct product previously defined is sound.

Proof
The proof follows immediately from the definition of the operators of the direct product. □

3.4.1 Running example

Consider the running example we discussed in Sect. 2.1. This example allocates two objects, a Talker instance at line 24 of Listing 1, and a Listener instance at line 16 of Listing 2. We represent these heap identifiers by $(\mathsf{Talker}, 24)$ and $(\mathsf{Listener}, 16)$, respectively. The allocation-based heap analysis $\overline{\mathbb{H}}$ then tracks the references to self in the constructor and methods of Talker, and Listener points to objects allocated at those program points. Instead, the string analysis $\overline{\mathbb{SV}}$ infers on Talker that (i) the constant talker is passed to the constructor of Node at line 8, and (ii) the name of the topic passed to method create_publisher at line 10 is the constant chatter. Similarly, its application to Listener infers that (i) the constant listener is passed to the constructor of Node at line 8, and (ii) the name of the topic passed to method create_subscription at line 9 is the constant chatter.

Having established a high-level formalization of the abstract semantics, we now move to its practical application in Sect. 3.5. In the next section, we take the abstract definitions and formalization principles presented thus far and apply

Table 1 ROS 2 APIs

Description	ROS 2 API method
Create a node	(1) rclpy.node.Node.__init__(name, namespace)
	(2) rclpy.create_node(name, namespace)
Create a topic publisher	(3) rclpy.node.Node.create_publisher(msg_type, topic)
Create a topic subscriber	(4) rclpy.node.Node.create_subscription(msg_type, topic)
Create a service server	(5) rclpy.node.Node.create_service(srv_type, srv_name)
Create a service client	(6) rclpy.node.Node.create_client(srv_type, srv_name)
Create an action server	(7) rclpy.action.ActionClient.__init__(node, action_type, action_name)
Create an action client	(8) rclpy.action.ActionServer.__init__(node, action_type, action_name)

them concretely to the task of extracting the computational graph from ROS 2 programs. This shift from theory to practice enables the generation of policies that adhere to the previously defined security principles, specifically focusing on automating access control through accurate computational graph extraction.

3.5 Extraction of the ROS computational graph

Starting from the results obtained by the static analysis we just formalized, we build up the ROS Computational Graph described in Sect. 2. In particular, we expect the analysis to provide an abstract for each statement $\mathsf{st} \in \mathsf{Statements}$ of our program. With an abuse of notation, we will denote the abstract state before at statement st by $(\overline{\mathsf{s}}_{\mathsf{st}}, \overline{\mathsf{h}}_{\mathsf{st}}) \in \overline{\mathbb{A}}$.

3.5.1 Definition

Nodes The ROS Computational Graph is a weighted graph with two different types of nodes:

- robotic nodes, representing a robot with its name and namespace, and
- communication channels, representing different ways of communication. In particular, ROS 2 allows communication through topics, services, and actions.

The following definitions formalize all these components.

Definition 5 (Robotic nodes)
Robotic nodes RobNode are represented as a name in Str and a namespace also in Str. Formally, $\mathsf{RobNode} = \mathsf{Str} \times \mathsf{Str}$.

Definition 6 (Communication channels (Topics, Services, and Actions))
Topics, Services, and Actions are represented by a name. Formally, $\mathsf{Topic} = \mathsf{Str}$, $\mathsf{Service} = \mathsf{Str}$, and $\mathsf{Action} = \mathsf{Str}$.

Definition 7 (ROS Computational Graph nodes)
ROS Computational Graph nodes can be robotic nodes or communication channels. Formally, $\mathsf{N} = \mathsf{RobNode} \cup \mathsf{Topic} \cup \mathsf{Service} \cup \mathsf{Action}$.

Edges In ROS Computational Graphs, edges can go (i) from robotic nodes to communication channels or (ii) from communication channels to robotic nodes. This means that direct communications among robotic nodes or communication channels are impossible. Edges are weighted with the type of data that is exchanged. The ROS 2 Python library represents this with a reference to a type. For the sake of simplicity, our analysis represents the type through the string representation of its name. In this way, we apply string constant propagation analysis to identify the type of exchanged data. Therefore, we represent by $\mathsf{ET} = \mathsf{Str}$ the type of communication. The following definition formalizes this.

Definition 8 (Edges)
The set of weighted edges W is defined as follows:

$$
\begin{aligned}
\mathsf{W} = {} & \mathsf{RobNode} \times \mathsf{Topic} \times \mathsf{ET} && \text{(Topic publisher)} \\
& \cup\ \mathsf{Topic} \times \mathsf{RobNode} \times \mathsf{ET} && \text{(Topic subscriber)} \\
& \cup\ \mathsf{RobNode} \times \mathsf{Service} \times \mathsf{ET} && \text{(Service client)} \\
& \cup\ \mathsf{Service} \times \mathsf{RobNode} \times \mathsf{ET} && \text{(Service server)} \\
& \cup\ \mathsf{RobNode} \times \mathsf{Action} \times \mathsf{ET} && \text{(Action client)} \\
& \cup\ \mathsf{Action} \times \mathsf{RobNode} \times \mathsf{ET} && \text{(Action server).}
\end{aligned}
$$

3.5.2 Extraction

Table 1 reports the main methods of the ROS 2 Python APIs that create robotic nodes, topic publishers and subscribers, service servers and clients, and action servers and clients. We formalize a simplified version of the APIs, omitting all the irrelevant parameters and some minor implementation details (e.g., namespace has a default None value when creating a node). The official ROS 2 documentation (available at https://docs.ros2.org/latest/api/rclpy/) provides full details on those APIs.

In the rest of the formalization, we will omit the namespace of the methods and classes. Remember that $\overline{\mathsf{s}}_{\mathsf{st}}$ and $\overline{\mathsf{h}}_{\mathsf{st}}$ refer to the abstract state before statement st of the string constant propagation analysis and heap abstraction, respectively. Our definitions will assume to receive precise results

from these analyses. Otherwise, we assume that the analysis will fail.[8]

Given a program $\mathsf{p} \subseteq \mathsf{Statements}$ (where p contains all the statements of the program), we extract nodes and edges as follows.

Definition 9 (Node extraction)

$$\mathsf{N_p} = \bigcup_{\mathsf{st}\in\mathsf{p}} \left\{ \begin{array}{r} (n, ns) : \ \mathsf{st} = \mathsf{Node.\_\_init\_\_(x,y)} \\ \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{x}) = n \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{y}) = ns \end{array} \right\} \cup \quad (1)$$

$$\bigcup_{\mathsf{st}\in\mathsf{p}} \left\{ \begin{array}{r} (n, ns) : \ \mathsf{st} = \mathsf{create\_node(x,y)} \\ \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{x}) = n \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{y}) = ns \end{array} \right\} \cup \quad (2)$$

$$\bigcup_{\mathsf{st}\in\mathsf{p}} \left\{ \begin{array}{r} t : \mathsf{st} = \mathsf{x.create\_publisher(y,z)} \wedge \\ t \in \mathsf{Topic} \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{y}) = t \end{array} \right\} \cup \quad (3)$$

$$\bigcup_{\mathsf{st}\in\mathsf{p}} \left\{ \begin{array}{r} t : \mathsf{st} = \mathsf{x.create\_subscription(y,z)} \wedge \\ t \in \mathsf{Topic} \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{y}) = t \end{array} \right\} \cup \quad (4)$$

$$\bigcup_{\mathsf{st}\in\mathsf{p}} \left\{ \begin{array}{r} s : \mathsf{st} = \mathsf{x.create\_service(y,z)} \wedge \\ s \in \mathsf{Service} \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{z}) = s \end{array} \right\} \cup \quad (5)$$

$$\bigcup_{\mathsf{st}\in\mathsf{p}} \left\{ \begin{array}{r} s : \mathsf{st} = \mathsf{x.create\_client(y,z)} \wedge \\ s \in \mathsf{Service} \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{z}) = s \end{array} \right\} \cup \quad (6)$$

$$\bigcup_{\mathsf{st}\in\mathsf{p}} \left\{ \begin{array}{r} a : \mathsf{st} = \mathsf{ActionClient.\_\_init\_\_(x,y,z)} \wedge \\ a \in \mathsf{Action} \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{z}) = a \end{array} \right\} \cup \quad (7)$$

$$\bigcup_{\mathsf{st}\in\mathsf{p}} \left\{ \begin{array}{r} a : \mathsf{st} = \mathsf{ActionServer.\_\_init\_\_(x,y,z)} \wedge \\ a \in \mathsf{Action} \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{z}) = a \end{array} \right\}. \quad (8)$$

The right part of the formalization reports the number of the API in Table 1 it refers to.

To proceed with the formalization of edges' extraction, we first need to define a function that, given an abstract heap identifier, retrieves the robotic node it is associated with. Since a constructor can be invoked only once on a specific object, and method create_node returns a fresh node instance, we assume this can lead only to one node. If this is not the case (thus, it might lead to zero or many nodes), we let the extraction process fail.

The process of extracting a robotic node from an abstract heap identifier is straightforward and formalized as follows.

Definition 10 (Robotic node extraction)

$$\begin{array}{l} \mathit{extractNode} : \overline{\mathsf{HeapId}} \rightarrow \mathsf{RobNode} \\ \mathit{extractNode}(\overline{\mathsf{id}}) = (n, ns) \text{ where} \\ \quad \exists \mathsf{st} \in \mathsf{p} : \mathsf{st} = \mathsf{Node.\_\_init\_\_(x,y)} \wedge \\ \qquad \overline{\mathsf{h}}_{\mathsf{st}}(\mathsf{self}) = \overline{\mathsf{id}} \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{x}) = n \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{y}) = ns \\ \quad \vee \\ \quad \exists \mathsf{st} \in \mathsf{p} : \mathsf{st} = \mathsf{create\_node(x,y)} \wedge \\ \qquad \overline{\mathbb{H}}[\![\mathsf{st}, \overline{\mathsf{h}}_{\mathsf{st}}]\!] = (\overline{\mathsf{id}}, \overline{\mathsf{h}}') \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{x}) = n \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{y}) = ns. \end{array}$$

[8] We adopted this approach instead of producing very conservative graphs since these would lead to useless access policies stating that a node can access any channel. The experimental results will report the number of failures and discuss how this can be reduced with more complex analyses.

We are finally in a position to formalize how our approach extracts edges of the ROS Computational Graph. Like nodes' extraction, we formalize how this produces a specific edge for each statement invoking one of the APIs in Table 1. We recall the API number in the right part of the formalization.

Definition 11 (Edge extraction)

$$\mathsf{E_p} = \bigcup_{\mathsf{st}\in\mathsf{p}} \left\{ \begin{array}{l} (n, t, m) : t \in \mathsf{Topic} \wedge \\ \quad \mathsf{st} = \mathsf{x.create\_publisher(y,z)} \\ \quad \wedge \mathit{extractNode}(\overline{\mathsf{h}}_{\mathsf{st}}(\mathsf{x})) = n \\ \quad \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{y}) = t \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{z}) = m \end{array} \right\} \cup \quad (3)$$

$$\bigcup_{\mathsf{st}\in\mathsf{p}} \left\{ \begin{array}{l} (t, n, m) : t \in \mathsf{Topic} \wedge \\ \quad \mathsf{st} = \mathsf{x.create\_subscription(y,z)} \\ \quad \wedge \mathit{extractNode}(\overline{\mathsf{h}}_{\mathsf{st}}(\mathsf{x})) = n \\ \quad \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{y}) = t \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{z}) = m \end{array} \right\} \cup \quad (4)$$

$$\bigcup_{\mathsf{st}\in\mathsf{p}} \left\{ \begin{array}{l} (s, n, m) : s \in \mathsf{Service} \wedge \\ \quad \mathsf{st} = \mathsf{x.create\_service(y,z)} \\ \quad \wedge \mathit{extractNode}(\overline{\mathsf{h}}_{\mathsf{st}}(\mathsf{x})) = n \\ \quad \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{y}) = m \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{z}) = s \end{array} \right\} \cup \quad (5)$$

$$\bigcup_{\mathsf{st}\in\mathsf{p}} \left\{ \begin{array}{l} (n, s, m) : s \in \mathsf{Service} \wedge \\ \quad \mathsf{st} = \mathsf{x.create\_client(y,z)} \\ \quad \wedge \mathit{extractNode}(\overline{\mathsf{h}}_{\mathsf{st}}(\mathsf{x})) = n \\ \quad \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{y}) = m \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{z}) = s \end{array} \right\} \cup \quad (6)$$

$$\bigcup_{\mathsf{st}\in\mathsf{p}} \left\{ \begin{array}{l} (n, a, m) : a \in \mathsf{Action} \wedge \\ \quad \mathsf{st} = \mathsf{ActionClient.\_\_init\_\_(x,y,z)} \\ \quad \wedge \mathit{extractNode}(\overline{\mathsf{h}}_{\mathsf{st}}(\mathsf{x})) = n \\ \quad \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{y}) = m \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{z}) = a \end{array} \right\} \cup \quad (7)$$

$$\bigcup_{\mathsf{st}\in\mathsf{p}} \left\{ \begin{array}{l} (a, n, m) : a \in \mathsf{Action} \wedge \\ \quad \mathsf{st} = \mathsf{ActionServer.\_\_init\_\_(x,y,z)} \\ \quad \wedge \mathit{extractNode}(\overline{\mathsf{h}}_{\mathsf{st}}(\mathsf{x})) = n \\ \quad \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{y}) = m \wedge \overline{\mathsf{s}}_{\mathsf{st}}(\mathsf{z}) = a \end{array} \right\}. \quad (8)$$

3.5.3 Running example

We now extract the ROS Computational Graph on the running example introduced in Sect. 2.1 starting from the results of the abstract semantics discussed in Sect. 3.4.1.

First of all, the nodes extraction (Definition 9) returns the following three nodes:

- ('talker', '') through API (1) since the constructor of Node is invoked at line 8 of Talker with string constant talker as the name of the topic, and no namespace (that by default is empty),
- ('listener', '') through API (1) since the constructor of Node is invoked at line 8 of Listener with string constant listener as the name of the topic, and no namespace, and
- topic 'chatter' through API (3) because of method call Node.create_publisher at line 10 of Talker class, and API rule 4 because of method call Node.create_subscription at line 9 of Listener class.

Therefore, the set of nodes computed on the running example is $N_p = \{(\text{'talker'}, ''), (\text{'listener'}, ''), \text{'chatter'}\}$.

We then notice that the robotic node extraction function (Definition 10) returns:

- $(\text{'talker'}, '')$ when applied to $(\text{Talker}, 24)$ since method Node.__init__ is invoked on this heap identifier at line 8 of Talker class, and
- $(\text{'listener'}, '')$ when applied to $(\text{Listener}, 16)$ since method Node.__init__ is invoked on this heap identifier at line 8 of Listener class.

Relying on these results, the following edges are extracted through Definition 11:

- $((\text{'talker'}, ''), \text{'chatter'}, \text{'String'})$ because of method call Node.create_publisher (API (3) in Table 1) at line 10 of Talker class, and
- $(\text{'chatter'}, (\text{'listener'}, ''), \text{'String'})$ because of method call Node.create_subscription (API (4) in Table 1) at line 9 of Listener class,

Therefore, the edges of the ROS Computational Graph are the following:

$$E_p = \left\{ \begin{array}{l} ((\text{'talker'}, ''), \text{'chatter'}, \text{'String'}), \\ (\text{'chatter'}, (\text{'listener'}, ''), \text{'String'}) \end{array} \right\}.$$

These nodes and edges represent the graph in Fig. 1.

Similarly, when considering the more complex chatter_callback function reported in Listing 8, an additional topic node 'chatter2' is created because of API (3) of Definition 9 because of method call Node.create_publisher at line 4 of the aforementioned figure. Because of this method call, an additional edge $((\text{'listener'}, ''), \text{'chatter2'}, \text{'String'})$ is produced because of API (3) of Definition 11. In this way, the graph reported in Fig. 2 is produced.

3.6 Policy extraction

As described in Sect. 2.3.1, enclaves group nodes that share common functionalities or security requirements. However, implementing such clustering necessitates a high-level understanding of the application, which extends beyond the current scope of our analysis. Consequently, our approach is conservative; it generates policies where each enclave contains only a single node. This conservative strategy ensures that each node's permissions are precisely defined and minimizes the risk of inadvertent privilege escalation.

Once the ROS Computational Graph is generated, creating the access policy is straightforward: for each robotic node in the graph, permissions are granted for interactions with all its directly connected communication channels. It is worth noting that although generating the policy is the ultimate objective, the primary complexity and novelty of our approach lie in constructing the ROS Computational Graph. This distinction emphasizes that the core contribution of our work is in leveraging LiSA's static analysis capabilities to achieve an exhaustive, accurate graph that represents all interactions within a ROS 2 application. The policy generation, while essential, is thus a straightforward derivation from this computed graph. Interested readers can find the overall algorithm in the Appendix C.

In our approach, access control policies are designed to be minimal, meaning they adhere strictly to the Principle of Least Privilege (PoLP) by granting only the permissions necessary for the nodes' specified functionalities. Minimality ensures that each node accesses only the topics, services, or actions required for its operation, reducing the risk of overprivileged access that could compromise security.

4 LiSA4ROS2

In this section, we discuss LiSA4ROS2 [9, 10], our implementation of the formalization presented in Sect. 3. In particular, given a Python program using the ROS 2 library, LiSA4ROS2:

1. First applies the static analysis formalized in Sect. 3.4;
2. Then extracts the graph as formalized in Sect. 3.5. This step might fail because the static analysis results are not precise enough. In such a case, LiSA4ROS2 fails on the given program;
3. Finally, extracts the access policy as formalized in Sect. 3.6.

This section is structured as follows: first, we introduce the Library for Static Analysis (LiSA), which serves as the foundation for our analysis. Then, we present the architecture of LiSA4ROS2, detailing our approach to analyzing the `rclpy` library.

4.1 Library for Static Analysis (LiSA)

In this subsection, we introduce the Library for Static Analysis (LiSA) [11, 19],[9] the foundation that allows us to develop a customized, sound, and efficient analysis for ROS 2 applications. LiSA is a Java library that helps developers build sound static analyzers with ease. LiSA uses abstract interpretation [18] to reason about programs.

Similar to other static analysis tools such as MOPSA [20, 21], Infer [22], and Ikos [23], LiSA is built to analyze multiple programming languages using a unified analysis engine. At its core, LiSA employs a language-agnostic intermediate representation (IR) of programs [24]. This allows it to support a variety of languages through dedicated front-ends for

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

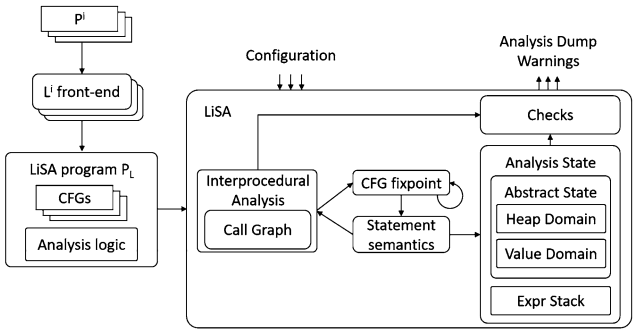

Fig. 4 LiSA architecture

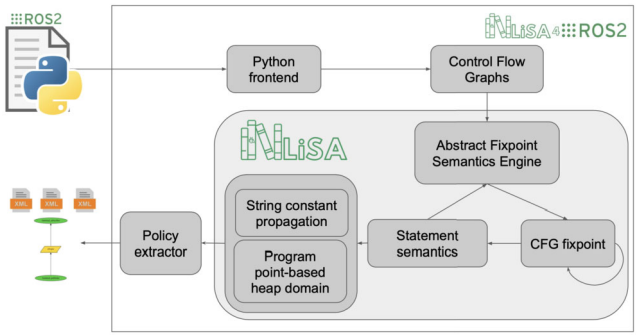

Fig. 5 LiSA4ROS2 architecture

Python, Rust, and Golang.[10] Being an open-source project, LiSA also supports custom front-ends, enabling developers to extend its capabilities to additional languages that are not officially supported.

LiSA's extensibility allows for enhanced precision in analysis by enabling the development of specialized domains tailored to specific application needs. Figure 4 illustrates the architecture of the LiSA library. The process begins with a program P written in a language L, which is transformed into an LiSA Program via a language-specific front-end. This LiSA Program comprises Control-Flow Graphs (CFGs) for each method encountered in the source code.

Each LiSA CFG models the flow of instructions within a method or function. These instructions are represented internally as LiSA Statements, which are generated by the front-end through parsing the source code. Each LiSA Statement encapsulates a specific operation or expression from the source code. The LiSA engine processes the LiSA Program, evaluating the program's behavior starting from the entry point using an interprocedural analysis approach. This analysis involves computing the fixpoint of a CFG based on the semantics of LiSA Statements.

LiSA's Abstract State framework, grounded in [25], models the program's memory through an abstract value domain (which tracks variable values) and an abstract heap domain (which monitors dynamic memory changes). Developers can leverage predefined abstract domains, such as those for constant propagation, shape analysis [26], or taint analysis [27, 28], or create custom domains to suit specific needs. To extract meaningful insights, such as warnings or code smells, users can employ checkers that iterate over the program's semantics to produce actionable results.

4.2 LiSA4ROS2 architecture

Figure 5 presents the architecture of LiSA4ROS2, illustrating the process flow. The tool ingests a set of ROS 2 Python source files into pyLiSA, the Python front-end for LiSA, which generates a corresponding set of LiSA programs. Each LiSA program consists of Control-Flow Graphs (CFGs). LiSA processes these CFGs by computing the semantics of the encountered statements, with special emphasis on calls to the rclpy library that we further discuss in Sect. 4.3. As mentioned earlier, we use a field-sensitive, point-based heap abstraction model combined with constant propagation to track the instantiation of nodes and the associated entity names.

Once the analysis is complete, we iterate semantically over all ROS 2-related statements to reconstruct the architecture of each node, leveraging the information retained in the abstract state. This process is repeated for every LiSA program, after which the collected information is consolidated into a shared object representing the communication graph. Once the network is constructed, we generate Access Control Policies by translating each node object in the network into an intermediate Java object that contains JAXB[11] annotations. These annotations facilitate mapping the POJO (Plain Old Java Object) to XML documents based on the SROS2 permission schema defined in the XSD file.

The generated SROS2 permissions must then be processed by the SROS2 CLI to produce the appropriate middleware-specific permissions. For better clarity and visualization, the tool exports the network data, including access control policies and other relevant statistics,[12] into an HTML file.

Currently, the tool is in its prototype stage and can analyze only Python files. However, the analysis can be extended to C++ sources.[13]

[10] All official front-ends are available here: https://github.com/orgs/lisa-analyzer.

[11] https://javaee.github.io/jaxb-v2/.

[12] Examples include lines of code, the number of nodes, topics, publishers per node, and global publisher counts.

[13] The complexity of analyzing applications built on the C++ ROS 2 library stems from the absence of a C++ front-end for LiSA. Building a front-end for any language is a labor-intensive and time-consuming process, which we have deferred as future work.

```
1 library rclpy:
2   location rclpy
3     class rclpy.node.Node:
4           instance method create_publisher:
                it.unive.pylisa.libraries.rclpy.node.CreatePublisher
5             libtype rclpy.publisher.Publisher*
6             param self libtype rclpy.node.Node*
7             param msg_type type
                it.unive.lisa.program.type.StringType::INSTANCE
8             param topic type
                it.unive.lisa.program.type.StringType::INSTANCE
9             param qos_profile type
                it.unive.lisa.type.Untyped::INSTANCE
```

Listing 12 SARL specification for the `rclpy.node.Node.create_publisher` method

4.3 rclpy semantics

As introduced in Sect. 2.2, the rclpy library serves as a Python wrapper around a C library. Internally, rclpy utilizes optimized code and specific C bindings that are challenging to analyze statically. Since our analysis is based solely on the application code, we lack access to the internal semantics of the rclpy library during the analysis phase, making a direct interprocedural analysis infeasible. To address this, we model the semantics of such libraries within LiSA by defining them as custom statements. This is accomplished using LiSA's Static Analysis Rule Language (SARL). We begin by specifying rules in a dedicated file that guides the analysis on how to interpret specific calls to rclpy. For illustration, Listing 12 provides a snippet written in SARL, demonstrating how these rules are articulated.

The SARL specification is parsed before the semantics computation begins. This snippet registers a class unit named `rclpy.node.Node` with an instance method `create_publisher` in the program under analysis. In LiSA, class units represent classes, while instance code members model method calls. Rather than extracting method semantics directly from the source code, SARL allows us to define the method's behavior programmatically using LiSA's statements.

In the snippet above, line 4 specifies that the semantics of the `rclpy.node.Node.create_publisher` instance method is defined in the Java class `it.unive.pylisa.libraries.rclpy.node.CreatePublisher`. The SARL snippet also outlines the method's signature, including the return value (line 5) and the number and types of arguments (lines 6–9). Specifically, the method being modeled returns an `rclpy.publisher.Publisher*`, where the asterisk indicates a pointer to an object in the heap. The `libtype` keyword signifies that the object's definition is provided using SARL.

The first parameter of the method is the object on which the method is called, with type `rclpy.node.Node`. The remaining three parameters include the message type (modeled as a `String`), the topic name (also a `String`), and the `qos_profile`, which does not have a defined type. Modeling the message type as a `String` simplifies the analysis

```
1 public <A extends AbstractState<A>> AnalysisState<A>
      forwardSemanticsAux(
2 InterproceduralAnalysis<A> interprocedural,
3 AnalysisState<A> state,
4 ExpressionSet[] params,
5 StatementStore<A> expressions)
6 throws SemanticException {
7       AnalysisState<A> result = state.bottom();
8
9       params[2] = SemanticsHelpers.nameExpansion(this,
              getSubExpressions()[0], params[2],
              interprocedural, state,
10      expressions);
11
12      String messageType =
              params[1].iterator().next().toString();
13      Constant c = new Constant(StringType.INSTANCE,
              messageType, getLocation());
14      params[1] = new ExpressionSet(c);
15
16      PyClassType publisherClassType =
              PyClassType.lookup("rclpy.publisher.Publisher");
17
18      PyNewObj publisherObj = new PyNewObj(this.getCFG(),
              (SourceCodeLocation) getLocation(), "__init__",
19      publisherClassType,
              Arrays.copyOfRange(getSubExpressions(), 1,
              getSubExpressions().length));
20      AnalysisState<A> newPublisherAS =
              publisherObj.forwardSemanticsAux(interprocedural,
21      state, Arrays.copyOfRange(params, 1, params.length),
              expressions);
22
23      result = result.lub(newPublisherAS);
24
25      return result;
26 }
```

Listing 13 Semantics of the `rclpy.node.Node.create_publisher` method

by focusing only on the type's name rather than the complex class structure in ROS 2. The `qos_profile` parameter, which defines the quality of service for the publisher, is set to an undefined type. This is acceptable since Python is untyped, and SARL does not support defining a parameter with multiple types.

LiSA processes the SARL specification to create a Class Unit for each class and a Code Unit for each method within those classes. During LiSA's interprocedural analysis, these units are used to resolve method calls by linking them to their defined semantics. Whenever the analysis encounters an unresolved method call, it queries all Code Units in the program to find a matching method signature. For the specific example at hand, when a method call on a `Node` object is encountered, LiSA searches the `rclpy.node.Node` unit for a matching Code Member. If a match is found, the call is registered in the Call Graph, and its semantics are computed based on the definition provided in the SARL, as illustrated in Listing 13.

The `create_publisher` method will instantiate a ROS 2 publisher in memory. In our analysis, this is modeled within the Abstract State by creating an object in the Heap Domain, accomplished through the appropriate semantics defined in the `PyNewObj` class.

```
1private ConstantPropagation ROSTopicNameExpansion(
2       ConstantPropagation left,
3       ConstantPropagation middle,
4       ConstantPropagation right,
5       ProgramPoint pp) {
6       if (left.isTop() || middle.isTop() || right.isTop())
7             return top();
8       String topicName = left.as(String.class);
9       String namespace = middle.as(String.class);
10      String nodeName = right.as(String.class);
11      // 1. remove prefix
12      if (topicName.startsWith("rostopic://")) {
13            topicName.replace("rostopic://", "");
14      }
15      // 2. fix namespace
16      if (!namespace.startsWith("/")) {
17            namespace = "/" + namespace;
18      }
19      // 3. substitute ~
20      topicName = topicName.replace("~",
            !namespace.equals("/") ? (namespace + "/" +
            nodeName) : "/" + nodeName);
21
22      // 4. check if we can prepend the namespace on
            topicName
23      if (!topicName.startsWith("/") &&
            !topicName.startsWith("{namespace}") &&
            !topicName.startsWith("{ns}")) {
24            topicName = !namespace.equals("/") ? (namespace
                  + "/" + topicName) : ("/" + topicName);
25      }
26      // 4. apply substitution
27      topicName = topicName.replace("{namespace}",
            namespace).replace("{ns}",
            namespace).replace("{node}", nodeName);
28      return new ConstantPropagation(new
            Constant(StringType.INSTANCE, topicName,
            pp.getLocation()));
29}
```

Listing 14 Topic name expansion

To create a publisher, we first need to expand its name (line 9). In ROS 2, the topic name is prefixed by the namespace of the node that defines it, if present. We replace the original name with the expanded name (parameter `params[2]`). Next, we retrieve the message type. For our purposes, we track only the name of the message classes, so we define a constant, and we assign it the string value of the message type (lines 12–13) and override the message type parameter accordingly. On line 18, we create a new Publisher using `PyNewObj`, passing all modified parameters of the `create_publisher` method except `self`.

For the expanded name, we replace the actual topic name with an expression. We model name expansion as an operation that takes three arguments (the topic name, the node name, and the namespace of the node) and produces the expanded name. The actual expansion is computed in the Constant Propagation domain (Listing 14). The process is straightforward: we simulate how ROS 2 performs name expansion by manipulating the string values of parameters. First, we remove the prefix `rostopic://` from the name if present. Then, we ensure that the namespace starts with a `/`. Next, we prepend the node's namespace to the topic name if the topic name is not absolute (i.e., it does not start with a `/`), and then we replace wildcards as follows:

- If the topic name contains {namespace} or {ns}, we replace it with the namespace.
- If the topic name contains {node}, we replace it with the node name.

5 Evaluation

To evaluate LiSA4ROS2, we created a dataset in November 2023 consisting of 682 GitHub repositories containing Python ROS 2 source code. This dataset is open source and can be accessed at https://zenodo.org/records/10817418.

From these repositories, we extracted all Python files that utilize the `rclpy` library and include calls related to `rclpy` for instantiating ROS 2 nodes. This filtering process yielded 5936 files. We processed these files using the Python frontend for LiSA, which successfully parsed 5552 files (93.53%), while 384 files (6.47%) encountered parsing errors.[14]

The parsed files were then analyzed through LiSA's interprocedural analysis to compute the semantics of the statements. LiSA was able to process 92.6% of these files (5141), while 7.4% (411) did not reach a fixpoint. Of the successfully processed files, meaningful results were obtained for 3406 files (66%). The remaining 1735 files (34%) presented cases where the analysis failed to compute precise names. Upon investigation, we found that these source entities' names were defined in external configuration files, which are not currently analyzed by our tool, but are a potential area for future work.

Table 2 presents some results of our analysis. In particular, from the complete set of analyzed examples, we selected those with notable features or complexities. The first four rows relate to the official ROS 2 examples,[15] which are minimal and demonstrate various ways to define ROS 2 nodes using the `rclpy` library. Our analysis successfully extracted all entities from the source code and produced valid policy files. The remaining rows represent a set of realistic Python ROS 2 applications with relatively complex ROS Computational Graphs that LiSA4ROS2 effectively analyzed. A notable example is the virtuoso repository, which exemplifies the potential scale and complexity of ROS 2 systems. This repository includes 52 nodes interconnected through a highly complex graph comprising 95 topics, 132 publishers, and 83 subscriptions. The ability to analyze such a vast and intricate network underscores the significance of a static analysis approach such as the one we propose with LiSA4ROS2. A runtime inspection would have required initiating and inspecting all 52 nodes to extract this information. Instead, with LiSA4ROS2, we can automatically have a comprehensive

[14] Details about these errors are available in the LiSA4ROS2 repository, under the analysis folder.

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

Table 2 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_min_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-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_tel_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

Table 3 LiSA4ROS2 execution time for the selected examples

PROJECT	Parsing time (ms)	Analysis time (ms)	Total time (ms)
pubsub_minimal	1369	263	2291
pubsub_min_oldschool	1380	284	2344
services_minimal	1575	304	2608
actions_minimal	1848	319	2910
mechaship	2057	1822	4587
solar-ros	1596	666	2976
ROS-LLM	1758	849	3467
Catch2023_hichewns	2804	3745	7710
fruit_collectors	1379	1253	3374
spatial-teleoperation	1494	501	2745
ProjectMarch	2131	1402	4394
5g_drone_ROS2	1947	971	3607
Virtuoso	9675	5946	16,585
eml4842_gps_nav	2156	1186	4212
MARV-ROS	2460	2059	5302
module89	4004	5392	10,199
zumopi_tel_system	2923	3134	6836

view of the entire system's communication and interaction patterns. This capability is particularly crucial for such large-scale systems where dynamic inspection would inevitably be time-consuming and resource-intensive. With this approach, we enable more efficient policy generation and security analysis, contributing to robust and scalable ROS 2 application development and deployment.

All the analyses was performed using a 2019 MacBook Pro 16, with a 32 GB 2667 MHz DDR4 RAM, and a 2.6 GHz Intel Core i7 6 core processor. Table 3 shows the execution time (in milliseconds) for each analysis. The second col-

umn details the time taken by the analyzer to parse all node sources, process the SARL file, and generate the corresponding LiSA program for each node. The third column indicates the time spent during the analysis phase, including network extraction. The last column displays the total time, encompassing both parsing and analysis times, along with a small overhead for generating HTML output from a predefined template. Parsing time is influenced by (i) the lines of code and (ii) the number of sources, while analysis time depends on (i) the number of rclpy calls in each node and (ii) the complexity of semantic computations (e.g., loops require more processing than simple statements). For the Virtuoso project, LiSA4ROS2 required a total of 16.585 seconds to complete the analysis. This included 9.675 seconds spent in the parsing phase and 5.946 seconds in the analysis phase.

6 Related work

To the best of our knowledge, no existing tools currently address the specific task we aim to undertake with LiSA4ROS2. While analogous work can be found in the literature, these approaches are typically applied in different domains and contexts. For example, in [29], the authors used static analysis to extract database interactions in web applications automatically. Similar to LiSA, their approach involved constructing a Control Flow Graph (CFG), but in a more simplified form known as an Interaction Flow Graph (IFG). However, our method is more comprehensive, as their analysis focuses solely on database interactions, whereas LiSA4ROS2 tracks communications via the publish-subscribe pattern used in ROS 2.

Our objectives more closely resemble those in [30], where statically extracted authorization graphs were employed in web applications to enforce Role-Based Access Control (RBAC) on resources. They used these graphs to identify vulnerabilities, validate policies, re-document policies, conduct role mining, and simplify authorization data. Similarly, we aim to achieve comparable outcomes, but within the complex multitiered architecture of robotic systems, where multiple nodes access resources through publish–subscribe mechanisms spanning various domains. Although prior work has applied static analysis to ROS, it differs from our specific goals. In ROS 1, notable efforts include HAROS [31] and the related analyses found in [32], but it is important to note that these tools do not support ROS 2, and there is no indication that such support is forthcoming [33].

In the context of ROS 2, [34] focused on formally verifying the Data Distribution Service (DDS) component in ROS 2, providing an abstraction and formalization of DDS based on probabilistic timed automata. Their objective was to verify properties like security (no deadlock), liveness (ensuring a node can reach send_wait), and priority (higher-priority nodes sending data first). Although their formal verification is insightful, it differs significantly from the context and goals of our research. Other approaches have concentrated on Runtime Verification (RV), where developers can validate the behavior of safety-critical systems that are too complex for formal verification. However, creating RV monitors is challenging, and errors in these monitors could jeopardize the entire system. The authors in [35] present a formal approach for generating runtime monitors for ROS 2 applications using the Formal Requirement Elicitation Tool (FRET) and the Ogma integration tool. Their focus, however, is on incorporating ROS 2 packages into larger systems, which falls outside the scope of our work.

Several RV efforts are worth mentioning, but they support only ROS 1 and lack compatibility with ROS 2. These include (i) ROSRV [36], with safety and security properties to be defined in a formal specification language and verified via automatically generated monitors, (ii) ROSMonitoring [37], which supports multiple ROS distributions and is agnostic to the specification formalism, and (iii) DeRoS [38], a domain-specific language and monitoring system designed specifically for ROS 1. Our approach, focused on enumerating ROS Computational Graph resources, seeks to refine the modeling phase described in the SROS2 paper [7]. Developers can inspect the graph using ROS 2 API command line tools or by employing *scapy* to dissect and decode DDS network packets [39]. Additionally, we can leverage recent improvements from [40], which extended `ros2_tracing` for real-time tracing of ROS 2 messages. Although originally intended to uncover causal links between input/output messages and indirect causal paths, incorporating this tool into our modeling analysis offers a valuable supplemental component.

However, despite the utility of these methods, they can explore only a limited subset of potential execution paths. Constrained by test execution conditions, such as specific input values and configurations, this limitation risks leaving parts of the system unexamined, which may result in misconfigured access policies. Consequently, hidden resources or malicious elements might persist undetected due to overly permissive rules [41].

Moreover, the Open Source Robotics Foundation (OSRF) developed a tool to translate the Node Interface Definition Language (NoDL) description of an ROS system into SROS2 policy [42]. However, as of this writing, development has stalled due to various issues with the definition and adoption of NoDL in ROS 2 [43].

7 Future work

Current limitations restrict LiSA4ROS2's ability to fully analyze ROS packages that use dynamic configurations, such

as external files for namespace remapping, or that distribute node instantiation across multiple source files. Expanding LiSA4ROS2's compatibility with diverse ROS 2 project layouts could significantly broaden its applicability and integration within the ROS 2 ecosystem.

A particularly promising area for enhancement involves supporting transitive analysis of ROS 2 launch files, which now often use Python-like syntax or static markup languages like XML and YAML. Launch files are essential in managing complex ROS 2 applications, allowing for dynamic configurations that influence runtime behavior. By incorporating launch file analysis, LiSA4ROS2 could better capture communication structures defined at launch, thereby enhancing the precision of generated security policies and adapting them to reflect the actual deployment settings.

Another planned extension is support for ROS 2 applications written in C++ through the development of an LLVM-based front end for LiSA. This addition would allow LiSA4ROS2 to process code written with the rclcpp library, opening up its analysis capabilities to a broader range of ROS 2 projects. Furthermore, we aim to implement Information Flow Analysis to detect and manage the declassification of private messages automatically. This feature would enable LiSA4ROS2 to help safeguard sensitive data flows across ROS 2 applications, reinforcing its role as a tool for secure and accurate policy generation.

In addition, future work could involve a comparative evaluation between LiSA4ROS2's static analysis approach and existing dynamic techniques used in ROS 2. A comparison with dynamic techniques would allow us to quantify the benefits and trade-offs in terms of completeness, coverage, and accuracy. This would be especially relevant to understanding how LiSA4ROS2's static overapproximation compares with the typically more scenario-bound insights obtained through dynamic analysis, providing insights into the strengths of each approach and identifying areas for potential hybridization.

8 Conclusions

In this paper, we discussed LiSA4ROS2, an innovative tool designed to automate the extraction of ROS Computational Graphs through static analysis. We explored the application of formal methods to illustrate how static analysis can effectively address potential errors resulting from improper policy configurations and manage the complexities of manually crafting accurate and maintaining security policies.

Our empirical assessment of LiSA4ROS2 demonstrated its capability to handle both simple and complex deployments using real-world codebases. This evaluation highlights the tool's practical utility and its potential to streamline the establishment of security policies in ROS 2 environments.

We also provided a critical examination of the current version's limitations and identified several promising avenues for future research and development. In particular, research on the determination of similarities between policies for shared enclaves represents a significant opportunity for advancement. This enhancement could involve analyzing node interactions to identify patterns or similarities that justify grouping nodes based on their roles and security needs in highly distributed deployments. By implementing such techniques, we could develop more efficient and manageable policies, offering a nuanced understanding of the application's structure and security requirements. This would not only streamline policy management but also improve the scalability and adaptability of security frameworks in complex systems.

Overall, this work contributes to a comprehensive understanding of LiSA4ROS2's current capabilities and highlights various opportunities for future enhancements. These advancements could further refine the tool and extend its applicability, ultimately contributing to more robust and adaptable security solutions in the field of software architecture.

Appendix A: DDS permissions

The policies generated by LiSA4ROS2, in conjunction with the SROS2 toolchain, are middleware-agnostic, meaning they are abstracted from the specifics of the underlying middleware. When generating security artifacts, SROS2 translates these policies into a format that the middleware in use can interpret. The default middleware in ROS 2 is `Prosima Fast DDS`, a widely adopted DDS (Data Distribution Service) implementation. As SROS2 does not currently support policy generation for non-DDS middleware, and at the time of writing, only DDS-based middleware is officially supported in ROS 2,[16] we provide in this appendix the DDS-compliant XML policy generated by SROS2 for the running example discussed in Sect. 3.5.3.

A.1 Mapping ROS 2 entities to DDS entities

ROS 2 entities are abstractions mapped to corresponding DDS entities at the middleware level. Specifically, an ROS 2 node is referred to as a *participant* within the DDS ecosystem. Since DDS operates exclusively through the Publish/Subscribe (Pub/Sub) communication pattern, services and actions in ROS 2 must be translated into publishers and subscribers at the DDS layer. For instance, an ROS 2 service server named `srv` is implemented using a subscriber (to handle incoming connections) on the DDS topic

[16] https://docs.ros.org/en/jazzy/Concepts/Intermediate/About-Different-Middleware-Vendors.html.

`/rr/srvRequest` and a publisher (to send the result back to the client) on the DDS topic `/rq/srvReply` (we will clarify the `/rq` and `/rr` prefixes shortly). A node that intends to use this service creates a service client in ROS 2, which is represented at the DDS level by a publisher on `/rr/srvRequest` (to send the request) and a subscriber on `/rq/srvReply` (to receive the response). In the case of actions, these are modeled through a combination of services and topics.

To distinguish ROS 2 entities at the DDS level, all DDS topics generated by ROS 2 are automatically prefixed with a namespace that identifies the type of ROS 2 entity using the topic. For example, the `/rt` prefix is used to denote a standard ROS 2 topic. Thus, an ROS 2 topic named `/chatter` is mapped to the DDS topic `/rt/chatter`. Similarly, the prefixes `/rq` and `/rr` are used for services, corresponding to the Request and Response topics, respectively.

A.2 DDS policy generated by SROS2 during artifact creation

The example SROS2 policy generated for the listener enclave is shown in Listing 15.

As shown, this policy allows connections to all topics (via the `rt/*` rule), making it insecure. The other topics refer to system entities, which are further explained in Appendix B.

A.3 Minimal DDS policies generated by SROS2 starting from the LiSA4ROS2 policy

Listing 16 presents the DDS policy generated by executing the following command:

```
rosdev@testbed:~/ros2\_ws$ ros2 security
      generate_artifacts -k keystore -p policy.xml
```

In this case, `policy.xml` is the ROS 2 policy generated by LiSA4ROS for the listener node (see Listing 18 for the complete version).[17] This policy is minimal: it avoids the use of wildcards (*) and includes all the necessary publishers and subscriptions for ROS 2 system services and topics, as well as the publisher to the `/chatter2` topic and the subscription to the `/chatter` topic.

Appendix B: ROS 2 internal topics and services

Under the hood, ROS 2 nodes utilize internal topics and services. These system-level resources enable an ROS 2 node to connect to the underlying network and allow external entities to manage its behavior by adjusting its internal parameters. A ROS 2 node leverages the following internal resources:

[17] Note that the `ros_discovery_info` permissions are listed separately from the others. This is how SROS2 extracts the permissions; however, combining them with the others does not pose any issues.

```
<?xml version="1.0" encoding="UTF-8"?>
<dds>
 <permissions>
  <grant name="/talker_listener/listener">
       <subject_name>CN=/talker_listener/listener</subject_name>
       <validity>
            <not_before>2024-08-20T14:37:20</not_before>
            <not_after>2034-08-19T14:37:20</not_after>
       </validity>
       <allow_rule>
         <domains>
              <id>0</id>
            </domains>
            <publish>
              <topics>
                   <topic>rq/*/_action/cancel_goalRequest</topic>
                   <topic>rq/*/_action/get_resultRequest</topic>
                   <topic>rq/*/_action/send_goalRequest</topic>
                   <topic>rq/*Request</topic>
                   <topic>rr/*/_action/cancel_goalReply</topic>
                   <topic>rr/*/_action/get_resultReply</topic>
                   <topic>rr/*/_action/send_goalReply</topic>
                   <topic>rt/*/_action/feedback</topic>
                   <topic>rt/*/_action/status</topic>
                   <topic>rr/*Reply</topic>
                   <topic>rt/*</topic>
              </topics>
            </publish>
            <subscribe>
              <topics>
                   <topic>rq/*/_action/cancel_goalRequest</topic>
                   <topic>rq/*/_action/get_resultRequest</topic>
                   <topic>rq/*/_action/send_goalRequest</topic>
                   <topic>rq/*Request</topic>
                   <topic>rr/*/_action/cancel_goalReply</topic>
                   <topic>rr/*/_action/get_resultReply</topic>
                   <topic>rr/*/_action/send_goalReply</topic>
                   <topic>rt/*/_action/feedback</topic>
                   <topic>rt/*/_action/status</topic>
                   <topic>rr/*Reply</topic>
                   <topic>rt/*</topic>
              </topics>
            </subscribe>
       </allow_rule>
       <allow_rule>
            <domains>
              <id>0</id>
            </domains>
            <publish>
              <topics>
                <topic>ros_discovery_info</topic>
              </topics>
            </publish>
            <subscribe>
              <topics>
                   <topic>ros_discovery_info</topic>
              </topics>
            </subscribe>
       </allow_rule>
       <default>DENY</default>
      </grant>
 </permissions>
</dds>
```

Listing 15 DDS dummy policy generated by SROS2

– `Service ~/describe_parameters` provides detailed information about one or more parameters, such as their types and descriptions.
– `Service ~/get_parameter_types` returns the types of specific parameters.
– `Service ~/get_parameters` retrieves the current values of specific parameters.
– `Service ~/list_parameters` lists all available parameters of the node.
– `Service ~/set_parameters` allows changing the values of one or more parameters.
– `Service ~/set_parameters_atomically` similar to `set_parameters`, but ensures that all parameter changes are applied at once.

Listing 16 DDS policy computed on the running example, using SROS2 with the LiSA4ROS2 generated policy

```
<?xml version="1.0" encoding="UTF-8"?>
<dds>
  <permissions>
    <grant name="/talker_listener/listener">
          <subject_name>CN=/talker_listener/listener</subject_name>
          <validity>
            <not_before>2024-08-19T17:39:32</not_before>
            <not_after>2034-08-18T17:39:32</not_after>
          </validity>
          <allow_rule>
            <domains>
                  <id>0</id>
            </domains>
            <publish>
              <topics>
                    <topic>rr/listener/get_type_descriptionReply</topic>
                    <topic>rr/listener/list_parametersReply</topic>
                    <topic>rr/listener/get_parametersReply</topic>
                    <topic>rr/listener/describe_parametersReply</topic>
                    <topic>rr/listener/get_parameter_typesReply</topic>
                    <topic>rr/listener/set_parameters_atomicallyReply</topic>
                    <topic>rr/listener/set_parametersReply</topic>
                    <topic>rt/chatter2</topic>
                    <topic>rt/parameter_events</topic>
                    <topic>rt/rosout</topic>
                  </topics>
            </publish>
            <subscribe>
                  <topics>
                    <topic>rq/listener/get_type_descriptionRequest</topic>
                    <topic>rq/listener/list_parametersRequest</topic>
                    <topic>rq/listener/get_parametersRequest</topic>
                    <topic>rq/listener/describe_parametersRequest</topic>
                    <topic>rq/listener/get_parameter_typesRequest</topic>
                    <topic>rq/listener/set_parameters_atomicallyRequest</topic>
                    <topic>rq/listener/set_parametersRequest</topic>
                    <topic>rt/chatter</topic>
                  </topics>
                </subscribe>
          </allow_rule>
          <allow_rule>
            <domains>
                  <id>0</id>
                </domains>
                <publish>
                  <topics>
                        <topic>ros_discovery_info</topic>
                  </topics>
                </publish>
                <subscribe>
                  <topics>
                        <topic>ros_discovery_info</topic>
                  </topics>
                </subscribe>
          </allow_rule>
          <default>DENY</default>
    </grant>
  </permissions>
</dds>
```

– Publisher to `parameter_events` tracks parameter changes.
– Publisher to `rosout` is used for logging and debugging purposes.

It is important to note that these services are node-specific (indicated by the ~prefix), while `parameter_events`, `rosout`, and `ros_discovery_info` are topics shared across all nodes. There is also a special topic called `ros_discovery_info`, which is hidden at the ROS 2 level and does not appear during dynamic network inspections using SROS2. Nodes use this topic to notify the underlying system of their presence and status within the network.

In Sect. 3.5.3, we have omitted internal topics and services for the sake of brevity and clarity. However, these are included in the policy generated by SROS2. The complete policy extracted by SROS2 is presented in Listing 17.

Since the names of these system-related entities are constants, LiSA4ROS2 includes these services and publishers in the generated policy. For completeness, Listing 18 displays the policy for the listener node in the running example, including the publisher to `chatter2`. Rather than using the private prefix ~, LiSA4ROS2 performs the expansion by replacing ~ with `/{namespace}/{nodename}`, as explained in Sect. 4.3.

Listing 17 Complete policy generated by SROS security tool for the Listener node

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

Listing 18 Complete policy generated by LiSA4ROS2 for the Listener node

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<policy version="0.2.0">
 <enclaves>
   <enclave path="">
        <profiles>
             <profile ns="/" node="listener">
               <topics publish="ALLOW">
                    <topic>/chatter2</topic>
                    <topic>/parameter_events</topic>
                    <topic>/rosout</topic>
               </topics>
               <topics subscribe="ALLOW">
                    <topic>/chatter</topic>
                </topics>
                <services reply="ALLOW">
                    <service>/listener/get_type_description</service>
                    <service>/listener/list_parameters</service>
                    <service>/listener/get_parameters</service>
                    <service>/listener/describe_parameters</service>
                    <service>/listener/get_parameter_types</service>
                    <service>/listener/set_parameters_atomically</service>
                    <service>/listener/set_parameters</service>
                </services>
              </profile>
         </profiles>
       /enclave>
 </enclaves>
</policy>
```

Appendix C: Extracting policy from a graph

Algorithm 1 reports a simplified version of the algorithm that LiSA4ROS2 uses to extract a policy from an ROS Computational Graph. While LiSA4ROS2 produces one policy file per node, the algorithm presented here produces one single policy file for all the ROS Computational Graph, with one node per enclave. We decided to present the algorithm in this way to simplify the explanation. This algorithm receives a graph derived from a program p represented by N_{p}, E_{p}. The beginning and end of the algorithm (lines 1–4 and 43–44) declare some standard tags of the XML access policy. The algorithm then iterates on all the robotic nodes (lines 5–42). For each node, it opens (lines 6–8) and closes (lines 39–41) the required tags for enclaves and policies. Then, for each different type of communication channel, the corresponding tags are added. In particular, for topic publishers, it adds a node `topics` where `publish` is allowed (opened at line 9 and closed at line 13). It then iterates over all the edges representing topic publishers (that are therefore in $\mathsf{RobNode} \times \mathsf{Topic} \times \mathsf{ET}$ as specified in Definition 8) and that refers to the current robotic node (by intersecting all the edges with $\{(n, ns)\} \times \mathsf{Topic} \times \mathsf{ET}$). For each edge, it simply adds a `topic` node with the topic name (line 11). The same process is applied to topic subscribers (lines 14–18), service clients (lines 19–23), service servers (lines 24–28), action clients (lines 29–33), and action servers (lines 34–38).

Algorithm 1 The algorithm that extracts the policy from the graph, where N_p, E_p represents the ROS Computational Graph produced on program p

```
1: print <?xml version="1.0" encoding="UTF-8"
2: print          standalone="yes"?>
3: print <policy version="0.2.0">
4: print <enclaves>
5: for all (n, ns) ∈ N_p ∩ RobNode do
6:     print <enclave>
7:     print <profiles>
8:     print <profile ns="ns" node=n>

       {Topic publisher}
9:     print <topics publish="ALLOW">
10:    for all ((n, ns), t, et) ∈ N_p ∩ ({(n, ns)} × Topic × ET) do
11:        print <topic> t </topic>
12:    end for
13:    print </topics>

       {Topic subscribers}
14:    print <topics subscribe="ALLOW">
15:    for all (t, (n, ns), et) ∈ N_p ∩ (Topic × {(n, ns)} × ET) do
16:        print <topic> t </topic>
17:    end for
18:    print </topics>

       {Service clients}
19:    print <services request="ALLOW">
20:    for all ((n, ns), s, et) ∈ N_p ∩ ({(n, ns)} × Service × ET) do
21:        print <service> s </service>
22:    end for
23:    print </services>

       {Service servers}
24:    print <services reply="ALLOW">
25:    for all (s, (n, ns), et) ∈ N_p ∩ (Service × {(n, ns)} × ET) do
26:        print <service> s </service>
27:    end for
28:    print </services>

       {Action clients}
29:    print <actions call="ALLOW">
30:    for all ((n, ns), s, et) ∈ N_p ∩ ({(n, ns)} × Action × ET) do
31:        print <action> a </action>
32:    end for
33:    print </actions>

       {Action servers}
34:    print <actions execute="ALLOW">
35:    for all (a, (n, ns), et) ∈ N_p ∩ (Action × {(n, ns)} × ET) do
36:        print <action a </action>
37:    end for
38:    print </actions>

39:    print </enclave>
40:    print </profiles>
41:    print </profile>
42: end for
43: print </enclaves>
44: print </policy>
```

C.1 Running example

We now apply Algorithm 1 on the running example introduced in Sect. 2.1. In particular, we apply this algorithm to the results computed in Sect. 3.5.3. Omitting the tags about services and actions (that are empty since no service or action is involved in the example), we obtain the policy reported in Listing 19. In particular, the opening and closing of the outer XML tags (lines 1–3 and 22–23 of the policy) correspond to lines 1–4 and 43–44 of the algorithm. The graph then contains two nodes. Therefore, the print statements at lines 6–8 and 39–41 of the algorithm produce lines 4–6 and 10–12 for Talker, and lines 13–15 and 19–21 for Listener in the policy. Finally, the algorithm (lines 9–13) allows the Talker to publish on topic chatter (lines 7–9 of the policy), and also (lines 14–18 of the algorithm) the Listener to listen to the same topic (lines 16–18 of the policy).

```
1<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <policy version="0.2.0">
3   <enclaves>
4     <enclave>
5       <profiles>
6         <profile ns="" node="talker">
7           <topics publish="ALLOW">
8             <topic>chatter</topic>
9           </topics>
10        </profile>
11      </profiles>
12    </enclave>
13    <enclave>
14      <profiles>
15        <profile ns="" node="listener">
16          <topics subscribe="ALLOW">
17            <topic>chatter</topic>
18          </topics>
19        </profile>
20      </profiles>
21    </enclave>
22  </enclaves>
23</policy>
```

Listing 19 Access policy computed on the running example

Intuitively, this policy corresponds to the union of the policies in Listing 5 and 6.

Instead, when considering the more complex code in Listing 8, inside the profile of the Listener the following additional publisher is produced:

```
1<topics publish="ALLOW">
2  <topic>chatter2</topic>
3</topics>
```

Funding This work was partially supported by SERICS (PE000000 14 – 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. Kirschgens, L.A., Ugarte, I.Z., Gil-Uriarte, E., Rosas, A.M., Vilches, V.M.: Robot hazards: from safety to security. CoRR (2018). arXiv:1806.06681
2. Caiazza, G., White, R., Cortesi, A.: Enhancing Security in ROS, pp. 3–15. Springer, Singapore (2019)
3. Mayoral-Vilches, V.: Robot cybersecurity, a review. Int. J. Cyber Forensics Adv. Threat Invest. **0**(0) (2022)
4. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y., et al.: ROS: an open-source robot operating system. In: ICRA Workshop on Open Source Software, vol. 3.2, p. 5. Kobe, Japan (2009)
5. Macenski, S., Foote, T., Gerkey, B., Lalancette, C., Woodall, W.: Robot operating system 2: design, architecture, and uses in the wild. Sci. Robot. **7**(66), eabm6074 (2022)
6. Mayoral-Vilches, V.: Robot cybersecurity, a review. Int. J. Cyber Forensics Adv. Threat Invest. (2022)
7. Mayoral-Vilches, V., White, R., Caiazza, G., Arguedas, M.: SROS2: usable cyber security tools for ROS 2. In: 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 11253–11259 (2022)
8. ROS2 core team: 2023-09 ROS 2 RMW alternate. https://discourse.ros.org/t/ros-2-alternative-middleware-report/33771
9. Zanatta, G., Caiazza, G., Ferrara, P., Negrini, L., White, R.: Automating ROS 2 security policies extraction through static analysis. In: Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (2024)
10. Zanatta, G., Ferrara, P., Lisovenko, T., Negrini, L., Caiazza, G., White, R.: 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, FTfJP 2024, pp. 5–10. Association for Computing Machinery, New York (2024)
11. Negrini, L., Ferrara, P., Arceri, V., Cortesi, A.: LiSA: A Generic Framework for Multilanguage Static Analysis, pp. 19–42. Springer, Singapore (2023)
12. Ferrara, P., Negrini, L., Arceri, V., Cortesi, A.: 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, pp. 1–6. Association for Computing Machinery, New York (2021)
13. Cousot, P., Cousot, R.: 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, pp. 238–252. ACM (1977)
14. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: 6th Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979, pp. 269–282. ACM Press (1979)
15. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. **13**(2), 181–210 (1991)
16. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994)
17. Ferrara, P.: Generic combination of heap and value analyses in abstract interpretation. In: Proceedings of VMCAI'14. LNCS. Springer, Berlin (2014)
18. Cousot, P.: Principles of Abstract Interpretation. MIT Press, Cambridge (2021)
19. Ferrara, P., Negrini, L., Arceri, V., Cortesi, A.: 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), SOAP 2021, pp. 1–6. ACM Press (2021)
20. Monat, R., Ouadjaout, A., Miné, A.: Static type analysis by abstract interpretation of Python programs. In: Proc. of the 34th European Conference on Object-Oriented Programming (ECOOP'20). Leibniz International Proceedings in Informatics (LIPIcs), vol. 166, pp. 17:1–17:29 (2020). Dagstuhl Publishing. http://www-apr.lip6.fr/~mine/publi/article-monat-al-ecoop20.pdf
21. Ouadjaout, A., Miné, A.: A library modeling language for the static analysis of C programs. In: Proc. of the 27th International Static Analysis Symposium (SAS'20). Lecture Notes in Computer Science (LNCS), vol. 12389, pp. 223–246. Springer, Berlin (2020). http://www-apr.lip6.fr/~mine/publi/ouadjaout-al-sas20.pdf
22. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at Facebook. Commun. ACM **62**(8), 62–70 (2019)
23. Brat, G., Navas, J.A., Shi, N., Venet, A.: IKOS: a framework for static analysis based on abstract interpretation. In: Proceedings, vol. 12, Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1–5, 2014, pp. 271–277. Springer (2014)
24. Zhang, B., Chen, W., Chiu, H.-C., Zhang, C.: Unveiling the power of intermediate representations for static analysis: a survey (2024)
25. Ferrara, P.: Generic combination of heap and value analyses in abstract interpretation. In: McMillan, K.L., Rival, X. (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 302–321. Springer, Berlin (2014)
26. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1999), pp. 105–118. ACM Press (1999)
27. Ernst, M.D., Lovato, A., Macedonio, D., Spiridon, C., Spoto, F.: Boolean formulas for the static identification of injection attacks in Java. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2015), pp. 130–145. Springer, Berlin (2015)
28. Spoto, F., Burato, E., Ernst, M.D., Ferrara, P., Lovato, A., Macedonio, D., Spiridon, C.: Static identification of injection attacks in Java. ACM Trans. Program. Lang. Syst. **41**(3), 18:1–18:58 (2019)
29. Ngo, M.N., Tan, H.B.K.: Applying static analysis for automated extraction of database interactions in web applications. Inf. Softw. Technol. **50**(3), 160–175 (2008)
30. Berger, B.J., Nguempnang, R., Sohr, K., Koschke, R.: Static extraction of enforced authorization policies SeeAuthz. In: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 187–197. IEEE Computer Society, Los Alamitos (2020)
31. Santos, A., Cunha, A., Macedo, N.: The high-assurance ROS framework. In: 2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE), pp. 37–40 (2021)
32. Santos, A., Cunha, A., Macedo, N., Lourenço, C.: A framework for quality assessment of ROS repositories. In: 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 4491–4496 (2016)
33. HAROS. To which extent is ROS 2 supported? https://github.com/git-afsantos/haros/issues/117 Accessed on 2024-02-28
34. Liu, Y., Guan, Y., Li, X., Wang, R., Zhang, J.: Formal analysis and verification of DDS in ROS 2. In: 2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE), pp. 1–5 (2018)
35. Perez, I., Mavridou, A., Pressburger, T., Will, A., Martin, P.J.: Monitoring ROS 2: from requirements to autonomous robots (2022). arXiv preprint. arXiv:2209.14030
36. Huang, J., Erdogan, C., Zhang, Y., Moore, B., Luo, Q., Sundaresan, A., Rosu, G.: Rosrv: runtime verification for robots. In: Proceedings, vol. 5, Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22–25, 2014, pp. 247–254. Springer (2014)

37. Ferrando, A., Cardoso, R.C., Fisher, M., Ancona, D., Franceschini, L., Mascardi, V.: ROSMonitoring: a runtime verification framework for ROS. In: Proceedings, vol. 21, Towards Autonomous Robotic Systems: 21st Annual Conference, TAROS 2020, Nottingham, UK, September 16, 2020, pp. 387–399. Springer (2020)
38. Adam, S., Larsen, M., Jensen, K., Schultz, U.P.: Towards rule-based dynamic safety monitoring for mobile robots. In: Proceedings, vol. 4, Simulation, Modeling, and Programming for Autonomous Robots: 4th International Conference, SIMPAR 2014, Bergamo, Italy, October 20–23, 2014, pp. 207–218. Springer (2014)
39. Rohith, R., Moharir, M., Shobha, G., et al.: Scapy – a powerful interactive packet manipulation program. In: 2018 International Conference on Networking, Embedded and Wireless Systems (ICNEWS), pp. 1–5. IEEE (2018)
40. Bédard, C., Lajoie, P.-Y., Beltrame, G., Dagenais, M.: Message flow analysis with complex causal links for distributed ROS 2 systems. Robot. Auton. Syst. **161**, 104361 (2023)
41. Deng, G., Xu, G., Zhou, Y., Zhang, T., Liu, Y.: On the (in)security of secure ROS 2. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS'22, pp. 739–753. Association for Computing Machinery, New York (2022)
42. OSRF. https://github.com/osrf/nodl_to_policy. Accessed on 2024-02-28
43. ROS2 Design. Design node interface definition language (IDL). https://github.com/ros2/design/pull/266. Accessed on 2024-02-28
44. Caiazza, G.: Application-level Security for Robotic Networks. PhD thesis, Ca' Foscari University of Venice, Italy (2021)