Decentralized Information Flow Control for the Robot Operating System

A PROJECT REPORT
SUBMITTED IN FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
Master of Technology(Research)
IN
Faculty of Engineering

BY

Chinmay Gameti

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

January, 2024

Declaration of Originality

I, **Chinmay V. Gameti**, with SR No. **04-04-00-10-22-19-1-17201** hereby declare that the material presented in the thesis titled

Decentralized Information Flow Control for the Robot Operating System

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2019-2022**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Advisor Signature

DEDICATED TO

My beloved parents

Whose unwavering support and guidance have been instrumental in shaping my journey. Their love and encouragement have been my constant pillars of strength, and I am deeply grateful for their presence in my life.

Acknowledgements

Abstract

The Robot Operating System (ROS) is a popular open-source middleware widely used in the robotics community. While ROS provides extensive support for robotic application development, it lacks certain fundamental security features, making ROS-based systems vulnerable to attacks that can compromise the application and user security. To address these challenges, ROS incorporates security plugins and libraries to protect against unauthorized access and ensure secure communication between ROS applications. However, these user-level security tools do not protect end-to-end information flow against operating system (OS)-level attacks.

This research introduces FlowROS, a decentralized information flow control (DIFC) system for ROS. FlowROS empowers ROS applications with fine-grained control over their sensitive information, providing a programmable interface and supporting explicit label propagation for modified ROS applications. FlowROS also leverages implicit label propagation for backward compatibility with unmodified ROS applications while guaranteeing end-to-end information flow control, including secrecy and integrity requirements. The implementation of FlowROS includes a kernel-level enforcement engine based on Linux security modules (LSM) to intercept sensitive communications within the system.

The contributions of this research include identifying the limitations of mandatory access control (MAC)-based policy frameworks in ROS, motivating the need for DIFC systems in robotics platforms, presenting FlowROS as a practical DIFC solution for ROS applications, addressing the inherent DIFC challenge in ROS, and demonstrating the robustness, security, and performance of FlowROS through case studies, evaluations, and practical policies.

Overall, FlowROS enhances the security of ROS-based systems by providing ROS applications explicit control over the flow of their sensitive information, mitigating vulnerabilities, and protecting against accidental data disclosure.

Publications based on this Thesis

Contents

List of Figures

List of Tables

LIST OF ABBREVIATIONS

ROS Robot Operating System

SROS Secure Robot Operating System

OMG Object Management Group

DDS Data Distribution Service

RTPS Real-Time Publish Subscribe

EDP Endpoint Discovery Protocol

PDP Participant Discovery Protocol

API Application Programming Interface

DAC Discretionary Access Control

MAC Mandatory Access Control

IFC Information Flow Control

DIFC Decentralized Information Flow Control

ACL Access Control List

TCB Trusted Computing Base

HTEE Hardware-base Trusted Execution Environment

RBAC Role-based Access Control

LSM Linux Security Module

Chapter 1

Introduction

1.1 Introduction

The Robot operating system **ROS** [57] is an open-source middleware that is heavily used and accepted in the robotics community. ROS is rapidly growing and widely used due to its support of software libraries and tools that are developer-friendly [58, 12, 57], providing ROS developers with a wholesome architecture for developing a wide variety of robotic applications [56, 26, 19]. ROS uses a publish/subscribe model where ROS nodes executing in separate process contexts or executing remotely on a distributed system can communicate with other ROS nodes by publishing/subscribing to the data using ROS `topics`. Topics are the communication abstractions ROS applications use to communicate with other ROS applications. For instance, Let us say a ROS node A publishes its data on the topic `UserData`; the ROS node that requires that piece of information (e.g., B) will subscribe to the topic `UserData` in order to receive the `UserData` feed. The earlier version of ROS (i.e., ROS version 1) middleware used master node architecture, such that the master node executing on the same ROS-based environment is responsible for establishing communication between ROS nodes using ROS topics. However, the master node architecture is susceptible to a single point of failure, leading the system to a crash and a non-functional state [23, 28]. ROS2, a recent version of ROS, integrates a Data Distribution Service (DDS) [5, 4, 11, 61, 6] that, by default, manages the communications between ROS applications and therefore mitigates the ROS-based environment from a failure (i.e., a single point of failure) and providing a reliable real-time publish/subscribe wired protocol [6, 10]. ROS2, by default, uses an open-source DDS implementation eProsima but still allows ROS developers to use any available open-source DDS libraries suitable for their use case. Henceforth, in this thesis, the word "ROS." will imply ROS2 unless explicitly mentioning ROS

version 1.

ROS provides better middleware support for robotic application development due to its diverse support for various robots and industrial use cases. However, it lacks some basic security features, making ROS-based systems vulnerable to attacks that may violate security for applications and users [54, 38, 21]. For instance, a compromised ROS application running on the system may try eavesdropping the communication between ROS applications. It can access or ex-filtrate the user's classified data and the application's private sensitive files. Also, a malicious application can publish or subscribe to topics it is not allowed to. To overcome these challenges, ROS provides libraries and service plugins to protect the system from such attacks [71, 16, 46] that prevent malicious applications from publishing (i.e., denial of service attacks, fault injection, etc.) or unauthorized subscription (i.e., sniffing), such that, the applications can mitigate such attacks by using Secure Robot Operating System (SROS), or secure DDS's authentication and access control service plugins [46]. Using SROS, applications can declare who can publish or subscribe to the topics using signed certificates, and the SROS layer will enforce it at run time [71]. ROS applications can monitor data flows using logging and data tagging plugins at the DDS layer. Note that eavesdropping on the packets can be mitigated using cryptographic plugins that the DDS layer by default does.

These are some user-land security tools that the ROS2 middleware supports, but the system is still far from perfect since ad-hoc security tools at the DDS layer do not guarantee end-to-end information flow protection on the system against OS-level attacks [49, 16, 46]. For instance, DDS primarily uses network sockets (i.e., `UDPV4` and `TCPV4`) to establish the connections and for actual pub/sub communications, but the capability exists for ROS applications executing on ROS environment to use other mediums of communications (e.g., shared memory (i.e.`msg_msg`), System V. (i.e.,`IPC`), and files). Using an OS-level interface exposes an attack surface to malicious applications to bypass the ROS layer completely and use OS-level artifacts to ex-filtrate sensitive information out of the system through a network or storage interface. To mitigate such vulnerabilities leading to accidental data disclosure of sensitive information, previous work [16] uses a centralized policy mechanism and kernel-level enforcement [3] to track the information flow of user and application's sensitive data not leaking out of the system. Similarly, past work [49] uses OS-managed security mechanisms (e.g., Controlled HW access using ports and SELinux, access control (SELinux), and Netfilters) to secure DDS-based systems (on robotics platform) from OS-level attacks. These frameworks leverage SELinux and AppArmor, which are Mandatory access control (Mandatory Access Control (MAC)) based systems. MAC systems provide OS-level information flow control guarantees by applying kernel-level enforcement of user or trusted system admin-given centralized security policies (i.e., MAC policies). Note

that MAC systems fundamentally suffer from two problems:

1. **Coarse granularity**. Since MAC systems apply process-level enforcement. The MAC-based system cannot distinguish different security classes of data at the process level. Therefore, they label the process with the highest security label associated with the data. (i.e., the process having data that belongs to the highest security class).

2. **Policy deduction**. Deducing a policy has always been challenging on MAC systems, which require system admin to thoroughly identify information flow in the system and create a system-wide centralized policy considering the secrecy and integrity requirements of the system that should cover all possible information flow path in the system [53, 69, 34, 35, 50].

Figure 1.1 below shows a scenario on a ROS-based system where an application publishes three types of data associated with three security classes (i.e., top-secret, secret, and nonsecret). Consider a scenario where a ROS-based system uses OS-level enforcement (SELinux/AppArmor) to guarantee end-to-end secrecy and integrity. The application wants to protect its data belonging to the security class "top secret," such that only specified trusted applications in the system are allowed to subscribe to this data. For the data belonging to the class "secret," any application can subscribe to this data. However, the data should not leave the system. The application does not want to put any restrictions on the flow of data belonging to the class "nonsecret," Such a requirement will force a MAC policy to classify the application to the highest security level that the data is associated with, therefore putting undue restrictions on all types of information flows that the application performs, i.e., information belonging to classes "secret" and "nonsecret" will suffer a similar information flow restrictions as of security class "top secret."

In operating systems, the natural evolution of the security mechanisms has highlighted these facts and tried rectifying the limitations that MAC systems impose [59]. These mechanisms have realized the need to segregate application data based on how applications see it or associate the security class with its data. Thus, it offers application developers a programmable interface that empowers them to control the flow of information.

We present **FlowROS**, an OS-level Decentralized Information Flow Control (DIFC) system for the robot operating system (ROS). FlowROS enables ROS applications by giving fine-grained control over their sensitive information using a programmable interface that the FlowROS exposes. FlowROS supports explicit label propagation for modified ROS applications [68, 44, 60, 33, 74] and leverages implicit label propagation mechanism to provide backward

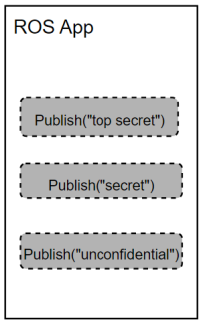


Figure 1.1: ROS motivating example

compatibility for unmodified ROS applications while still guaranteeing end-to-end information flow control (i.e., secrecy and integrity requirements). FlowROS supports using DIFC on the ROS platform by identifying the need for DIFC on the ROS platform and solving the critical challenge that the underlying ROS imposes (i.e., Label explosion[1]). We implement FlowROS from scratch and leverage Linux security modules (LSM) [72] for implementing the kernel-level enforcement engine that intercepts every sensitive communication in the system. We support our proposal with an open-source implementation.

[1]

To summarize our contributions:

- We identify the shortcomings of MAC-based policy frameworks on ROS and, therefore, motivate the use of the DIFC system on robotics platforms.
- We present FlowROS as a new DIFC system for ROS, empowering ROS applications to control their sensitive information from accidental data disclosures.
- We demonstrate a practical DIFC system, FlowROS, as a combination of explicit and implicit label propagation, making it backward compatible with unmodified ROS applications
- We identify and highlight an inherent DIFC challenge on ROS and provide a solution in FlowROS for making DIFC possible on the ROS platform.
- We present the robustness of FlowROS by demonstrating practical policies and a case study of an existing ROS application to use DIFC. We present a security evaluation of

[1] "Label explosion" refers to a situation where the entire system accumulates all distinct labels, resulting in a large restrictive label, leading the system to a non-functional state[44].

FlowROS on the NVidia Jetson TX2 board and a performance evaluation showing the overheads incurred at the application and system levels.

1.2 Outline

The further thesis is organized as follows:

- Chapter 2 gives an overview of ROS nomenclature, eProsima DDS, and Dynamic discovery protocol. And a detailed introduction to OS-level access control and information flow control mechanisms. We further present security and privacy concerns on ROS-based applications, defining our adversarial model and trusted computing base.
- Chapter 3 presents related work with respect to ROS and Information Flow Control (IFC) systems.
- Chapter 4 presents security and privacy-related work in the ROS framework and OS-level information flow control.
- Chapter 5 gives a detailed explanation of the implementation of FlowROS. The formal explanation of tags, labels, and label propagation rules in FlowROS. The Reference monitor implementation and FlowROS `ioctl` interface for ROS applications.
- Chapter 6 presents practical DIFC policies for ROS applications and evaluation of FlowROS on the NVIDIA Jetson TX2 board.
- In chapter 7, we finally conclude and address a few future directions.

Chapter 2

Background and Threat Model

This chapter gives background on Robot Operating System (ROS) middleware, which is extensively used in robotics platforms, and the eProsima Data Distribution Service (DDS) library that ROS uses for seamless communication between ROS applications. Then, we explain the evolution of classic OS-level security mechanisms from access control systems (i.e., Discretionary Access Control (DAC), MAC, etc.) to various decentralized information flow control systems (i.e., OS-level DIFC, PL-based DIFC, etc.).

2.1 ROS Framework

Robot Operating System (ROS)[2, 12, 57, 58] is a widely used middleware for robotics platforms. In recent years, it has gained popularity in the robotics community. ROS enables applications executing on various kinds of robots, autonomous vehicles, and drones to quickly implement a rich set of functionalities, including those for 2D and 3D simultaneous localization and mapping, navigation, and perception. Many robotics platforms are distributed, and ROS enables applications to work seamlessly across a set of coordinating robots. For example, ROS-based applications are used to control a group of collaborating robots on a factory shop floor or a swarm of drones together accomplishing a task. Many industries are shifting to a networked-based environment to increase production by automating many tasks. The ROS middleware is expected to grow and have applications on many different platforms [23].

2.1.1 Nomenclature

ROS's nomenclature comprises four fundamental concepts: nodes, topics, messages, and services [52].

- A node in ROS is an entity that performs computation. Nodes execute at the granularity of a process in operating systems. The process-level granularity of ROS nodes enables ROS for a modular design, which works similarly to software modules. As a result, in the ROS environment, the terms nodes and software module are often interchangeable.

- In ROS, Nodes can communicate with other nodes by passing messages. The messages are nothing but a typed data structure used for communication. Some primitive types of messages include (integer, boolean, character, floating-point, etc.). Also, messages can further be composed of many other different types and different primitive types of messages.

- Nodes communicate with each other by passing messages as the ROS middleware library provides a simple abstraction to applications for data communication. Topics are nothing but a string used by the ROS layer. ROS provides a publish/subscribe-based system, and applications can create publishers and subscribers associated with topics. Applications can publish messages on topics that are then delivered to applications that have subscribed to those topics. Applications declare the set of topics they post or subscribe to using manifests. Then, the Data Distribution Service (DDS), part of ROS middleware, handles matchmaking and message delivery underneath using the interfaces given by OS. Multiple publishers and subscribers can concurrently exist at the same time on a single topic. Also, a node can publish/subscribe to many topics simultaneously.

- The publish/subscribe model is based on topics. It is a beneficial paradigm for applications running in the ROS environment. The broadcast nature of the message-passing protocol used in these models does not allow synchronous transactions, which some applications might need (e.g., web services-based applications). ROS also has service nodes that are of request and response types, and it has well-defined sets of requests and response messages.

ROS2 is redesigned from its previous version ROS1, so it becomes versatile for many robotic applications that can utilize the ROS environment in a better way. The earlier version of ROS had a notion of the centralized master node that managed communications among ROS1 nodes—keeping track of nodes that broadcast its publishers and subscribers and create communication links between them. Upon creating the communication link, publishers can send

messages directly to subscribers by publishing their data on topics, and subscribers can receive the data by subscribing to it. The failure of a central node can break down an entire system and lead to some DOS attacks [23, 24, 39, 28]. ROS2 comes up with an open-source middleware service, a decentralized data distribution model (Data Distribution Service) [5, 10, 6] developed for embedded and real-time systems; DDS allows applications to discover each other at runtime dynamically, known as dynamic discovery protocol.

2.1.2 eProsima DDS

Fast data distribution service, Fast DDS, also formerly known as Fast RTPS, is specifically designed for distributed applications. eProsima implements an easy-to-use data-centric communication medium in the form of middleware. ROS, by default, comes with eProsima middleware for its real-time publish-subscribe model. There are many pub/sub implementations of DDS, but eProsima is a widely used, efficient, and performance-oriented implementation[61, 4, 7, 13, 9]. We focus on the eProsima-based version of ROS for our FlowROS system. Figure 2.1 shows the architecture of eProsima implementation, specifying different components.
The concept of Domain is used as a communication medium in DDS. A domain acts as a distinct communication layer, enabling applications to seamlessly interact and discover each other's publishers and subscribers within a network. In this context, DDS introduces the concept of a DomainID, which serves to logically separate networks within the same physical infrastructure. This means that multiple ROS2 systems can operate concurrently on a single machine or within the same physical network without any mutual interference. Consequently, only those applications or nodes that share the same DomainID (i.e., belong to the same domain) are capable of discovering each other through the RTPS wired protocol of DDS. Essentially, a ROS system constitutes a communication graph comprising a collection of nodes, topics, and services facilitated by DDS. Each ROS system is linked to a specific DomainID, ensuring that all nodes within a system are initialized with the same DomainID. Users can set the DomainID either through environmental variables (e.g., using ROS_DOMAIN_ID) or programmatically during the development of a ROS application. Please note that any further discussions regarding Topics, DDS entities, and the Discovery protocol will be assumed to take place within the context of the same DDS domain.

Topics are fundamental to the discovery protocol and communication (i.e., data exchange) among ROS applications. Each DDS entity, such as publishers or subscribers within a ROS application, shares information about the topics it publishes or subscribes to with the DDS

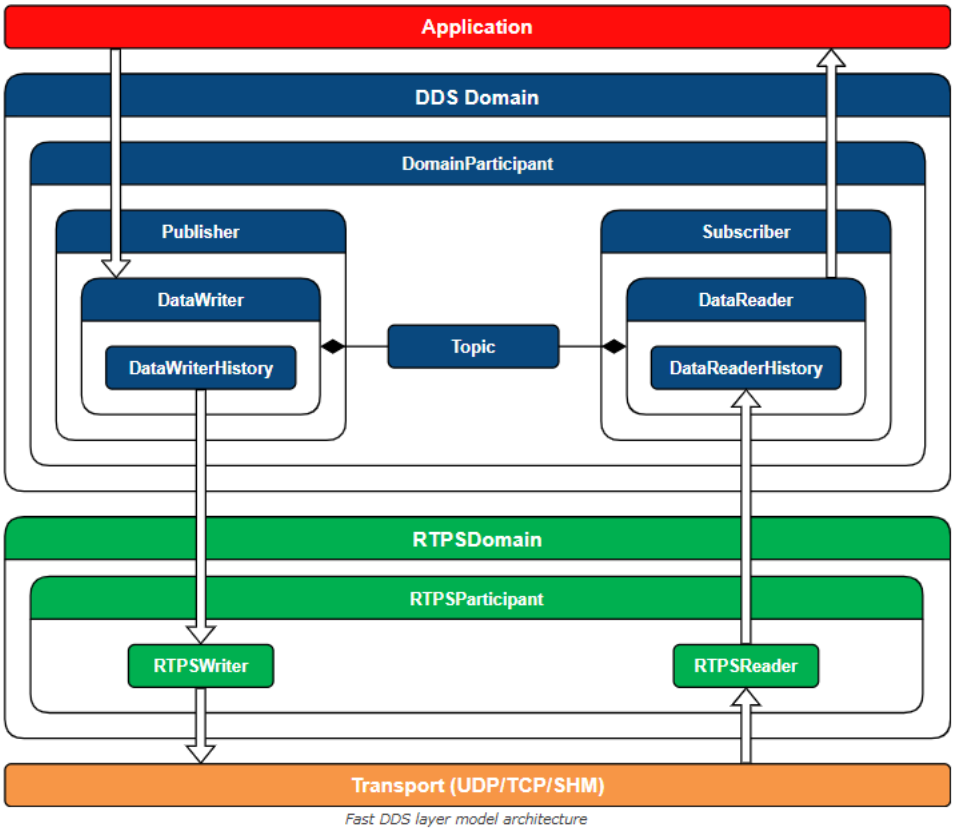


Figure 2.1: eProsima DDS architecture (Image borrowed from eProsima Fast DDS documentation [47, 7])

layer. This information includes the topic name, data type, Quality of Service (QoS), and other relevant details. Multiple topics can exist within a domain, and applications can publish or subscribe to these topics. The DDS middleware is responsible for routing messages based on topic names. When a publisher sends a message on a topic, the DDS middleware efficiently routes this message to the subscribers of that topic. It accomplishes this by keeping track of all active publishers and subscribers for each topic and creates communication channels only between the nodes that are relevant to the exchange. This approach helps the DDS middleware optimize network traffic, avoiding the need to create communication channels between all pairs of nodes. Since topic discovery is a part of the automatic discovery protocol in DDS, it is announced to all other entities in the system. However, this announcement, which involves metadata traffic, is managed by the DDS middleware and is not disclosed to the applications.

DataWriter: The publisher is a DDS entity that publishes messages on topics associated with the DataWriter entity. An application developer uses a high-level publish API to publish data. DataWriter, which is part of the publisher, has a Data-Writer history cache; Data published by the application is temporarily written to the Data-Writer cache, which is then later passed on to the Real-Time Publish Subscribe (RTPS) layer. All data writers are bound to a specific topic, and subscribers that match this topic will be able to receive the data.
DataReader: All DataReaders are bound to a specific topic for receiving data publication updates on that topic. Data readers are mainly associated with subscribers. DataReaders are created when applications create subscribers. DataReaders have a DataReader history cache that stores the data published or sent by a DataWriter on the topic that DataReader is associated with.
eProsima implements its DDS middleware mainly in two layers. The first layer is a DDS layer, a high-level abstract layer for applications using DDS; this layer is for managing data for the Publishers and Subscriber interface provided to applications before sending data over the communication channel. The second layer is the RTPS layer, which implements a lower level or internal layer for data communication. The protocol is based on the OMG group's RTPS standards [9, 10]. The API Domain layer has more control over the actual data communication over the wired protocol; it uses OS-level abstractions (such as sockets, shared memory, pipes, etc.) to manage data communication between RTPSWriters and RTPSReaders.

The final transport layer is where the actual data transfer is carried out over the physical communication medium between DDS entities. The architecture of DDS is made such that the middleware service can run on any API service beneath the ROS framework (i.e., OS-level

services). Some internal transport layers it uses are UDPV4, UDPV6, TCPV4, TCPV6, Shared memory (SHM), etc.

2.1.2.1 Dynamic Discovery Protocol

DDS enables applications to work seamlessly without the hassle of creating network communication with other running applications. This will require application writers to add support for managing communication links between applications upon initialization, destruction, and network failures. DDS makes this task of management easier for applications. In a ROS environment, applications do not need to know a priori where other applications reside, and they might be running on the same platform or remotely. The actual data transfer between applications happens through OS-level abstractions (i.e., sockets, files, shm). DDS uses a discovery mechanism to facilitate the easy and efficient management of data exchange protocol for ROS. It automatically detects active applications on the system and matches DataWriters with DataReaders.

1. **Participant Discovery Phase**
   Every application on the system initializes itself on a specific Domain by creating an instance of DomainParticipant class. That enables the DDS layer to listen to meta-traffic by other domain participants on the system and create communication links between them. Domain Participant sends announcement and acknowledgment messages periodically on the Domain. These messages consist of Uni-cast addresses (such as IP and Port) that the Domain participant listens to for meta-data traffic and data sent by other applications known as user-data traffic. The DDS layer uses meta-data traffic to manage communication among DomainParticipants. The acknowledgment messages relating to participant discovery are usually sent in a broadcast manner so that everyone listens to them. Furthermore, these messages are sent over a well-defined multi-cast address and port that are calculated using the DomainID.

2. **Endpoint Discovery Phase**
   DataReaders and DataWriters are part of Publishers and Subscribers of Domain participants. DomainParticipants send acknowledgment messages to each other to discover their endpoints over the communication link created by the Participant Discovery Protocol (PDP) protocol. Communication for the Endpoint discovery phase is usually of one-to-one uni-cast type, unlike the messages sent in the participant discovery phase using the broadcast method on well-defined ports.

OMG, a standards development organization, has given DDS interoperability wire protocol for real-time publish-subscribe. OMG standards specify mainly three layers: that are platform-independent module (PIM), platform-specific module (PSM), and serialized payload representation [10].

2.2 Access Control

2.2.1 Secrecy and Integrity

A process is an execution context for a program to perform computation on the system. Similar to process-level abstraction that protects other execution contexts executing on the machine, the Protection domain, which is used, also controls the access among processes belonging to different security contexts. The term protection domain is used to specify what a process can do, and In some sense, the protection domain represents a group or a security context. Each process belongs to some protection domain, which then can be used to monitor the access to system resources that are performed by the process. Sometimes due to bugs, some processes can inadvertently or maliciously impact the execution of another process by taking control over processes having bugs. Typically, A system uses some sparse data structure (e.g., Access Control List (ACL), Matrix) to keep the record of valid accesses or operations that a process can do for all the protection domains in the system. These protection systems' security guarantees are generally secrecy, integrity, and availability. Some important terms that are used to describe protection systems are subjects, objects, and operations. Processes and users belong to the subject, objects are OS-level resources (i.e., files, sockets, inodes), and operations are generally tasks (i.e., read, write, and execute) that subjects can perform on objects.
Secrecy and integrity are two primary security goals that protection systems can provide. In general, these systems aim for models of either secrecy or integrity. However, some unique access control systems focus on both [17, 18, 64]. Some objects may have sensitive or confidential information that should not be disclosed to unauthorized processes or subjects. Hence secrecy policy limits unauthorized subjects from accessing (i.e., malicious processes from performing read operations on sensitive kernel-level objects) containing sensitive information. Similarly, some objects may have information that other objects and subjects rely upon. Corrupted information of such sensitive data might break down other running processes or, as a result, the whole system. Hence, the integrity goal prevents corrupting sensitive data that others rely upon.

	File1	File 2	File 3
P1	r	r,w	r,w
P2		r	r,w

Figure 2.2: Access Matrix for DAC, P1 and P2 are processes and there access rights for file f1 and f2

2.2.1.1 Discreationary Access Control

In the DAC (Discreationary Access Control) system, permissions for subjects to access objects are determined based on policies. Various representations, such as the Lampson access matrix, access control lists (ACL), and capability lists, can explain the security of DAC. The access matrix is just a data structure representing subjects and objects, with each entry in the cell defining the set of allowed operations for a principal.

However, the issue with DAC arises from all objects' access privileges being left to the discreation of their users or processes. It allows any untrusted process to modify the protection state, posing a significant security risk. For instance, consider access matrix 2.2 file 1 containing sensitive information and process one having read access to it. If process 1 is benign but has a code vulnerability, malicious process two can exploit this vulnerability and leak the information via file 2. Since process 1 has read access to file 1, process 2 can also grant itself read access to file 1 by altering the data structure used for a DAC. This lack of control can lead to potential data leaks and unauthorized access. In conclusion, it is hard to reason about the security of the system based on the protection state it is in and all mutable protection states that can be derived. Hence for the DAC system, it is an undecidable problem known as a safety problem [27]. To protect the system completely, such that information can not leak, and any untrusted process should not be able to modify the protection state (access matrix). There is a need for an authorized system administrator who can only assign and modify the transition state, leading to a Mandatory Access Control system.

2.2.1.2 Mandatory Access Control

As mentioned earlier, the DAC system allows untrusted processes to modify the access matrix, leading to potential issues like information leaks and control hijacking of processes with vulner-

abilities. As a result, DAC falls short of guaranteeing secrecy and integrity requirements. To overcome these limitations, mandatory access control (MAC) systems have emerged.

The primary objective of MAC is to address the fundamental necessity for a protection system that can enforce secrecy and integrity requirements, even in the presence of malicious software within the system. In MAC, the protection state, represented by policies, must originate from a trusted and authorized source capable of identifying the secrecy and integrity needs of the system. This ensures a robust and reliable mechanism for maintaining the security of the system's data and processes.

MAC systems consist of three primary operations types delegated to a trusted system admin.

1. Labeling.
2. Mandatory protection state.
3. Transition(i.e., label change).

Labels serve as system-defined identifiers that assign specific security classes or semantics to subjects and objects within the system. These security classes associated with processes (or principals) and objects enable the OS-level reference monitor to enforce security policies dynamically during runtime based on these labels. A trusted administrator is responsible for assigning these labels to processes and objects following the security requirements of the applications or the entire system. Once labels are assigned, they remain immutable, ensuring the tamper-proof nature of the MAC system.

The labeling represents how subjects and objects in the systems are mapped with labels. Whenever a new file or process is created, it is assigned some label associated with a security classification. A mandatory protection 'state' shows a protection state that describes the set of allowed operations that subjects can perform on objects in the form of an access control list or access matrix. The access matrix shows relations among labels; these relations show a set of allowable and non-allowable operations that an OS-level reference monitor will use to enforce the security policies given by a system admin.

In operating systems, new processes are spawned through a fork and execve system calls. Upon calling the execve call, the process's binary file (i.e., code and data of a program) is replaced with the binary path given by the program (or application) at runtime, also known as policy loading. In such scenarios, transitions of the system admin-assigned labels become necessary. The new program may have different access rights than the previous one and should be confined by giving new labels meeting the security requirement.

2.2.1.3 Information Flow Control (IFC) Challenges

An Information flow control [22] policy prevents accidental data disclosure or provides secrecy by applying constraints on the data objects and subjects that are well-known in the system. An IFC system expects a system administrator to produce a set of well-defined rules for data propagation and release. Hence, it requires policies to be complete to avoid accidental data leaks. Considering we have a solid policy, an IFC (such as MAC) can then enforce that policy at the OS level to prevent such data leaks [43, 48, 16, 33]. A policy includes a set of allowable rules in the form of predefined security labels attached to these subjects and objects. The policy represents the ordering of security classes, determining how data will flow between any two security classes. A secrecy requirement will follow "no flow up" (i.e., lower to higher security class). And flows that are not allowed in the system should go by a declassification.

A centralized policy requires the system admin to analyze the whole system's provenance (i.e., audit logs) generated at runtime by a whole system provenance capture modules in the kernel. These capture modules record every read/write access in the system by subjects (i.e., processes) on objects. Then it is the responsibility of a system admin to analyze and prepare a policy thoroughly. For example, AppArmor, SELinux, Smack, etc., are some mainstream MAC systems that provide tools to ease the process of policy generation, such as auditd mode, etc. But coming up with a full-proof policy is still a complex problem [35, 50] because to protect the system from attacks and misbehavior, a policy should cover all most every allowable access, and audit logs should cover all possible access paths for all the applications running in the system. Further, there are attempts to solve this problem [35, 34, 53, 69, 70]; for example, there is a path-based confidence framework (such as ASPGEN for AppArmor, etc.) that help in policy generation by categorizing objects and increasing the code coverage.

2.2.1.4 Decentralized Information Flow Control (DIFC)

As discussed in the previous section, with the increase in the complexity of modern software and the need for software to secure sensitive data, it is challenging to express and enforce these security needs using standard DAC and MAC solutions. Lately, better security mechanisms have been developed for application developers to control their data, as in how their sensitive data will be consumed and propagated further to other applications on the system that may maliciously use it (i.e., SEApp [59]). As a natural evolution of security needs, DIFC enables application developers to express the categories of the data in the app by giving policies in the form of allowed or disallowed rules.

For instance, consider an application that has data of different security class (e.g., confidential, non-confidential, etc.). For the data tagged confidential, whichever process tries to read it will not be able to access any file or files not having the confidential tag to write. If a process or an application still wants to write, it needs to declassify itself by following the declassification principles. On the contrary, non-confidential information can be consumed and propagated without the need for declassification. There are mainly three types of DIFC models. One DIFC at the language level [41, 40, 42], the second at the operating system level [68, 74, 33], and the third at the architecture level [67, 75]. Where each of these approaches comes with its strengths and weaknesses [60].

Language-based DIFC. These DIFC systems use program analysis and data structures to follow secrecy and integrity constraints. It is more fine granular and static in nature as it attaches labels to the data structures and objects at the application program level; the primary need for a language-based DIFC is that it requires an intrusive system or a completely new language. That is the reason these models use JVM or JAVA language. This approach gives better control as it is enforced at the byte-code generation layer. But the disadvantage is that it trusts the operating system and cannot protect against OS-level object violations (i.e., files and sockets) [60, 40].

OS-level DIFC. OS-based DIFC systems support information flow tracking at the abstraction of objects and subjects in the system. These systems have a reference monitor as a part of the kernel layer, which is implemented using Linux Security Module (LSM) [72]. LSM provides some basic hooks in the Linux kernel code, which are then used to develop security rules. OS-based DIFC uses these hooks to implement their reference monitor that tracks the information flow. These systems works based on labels and capabilities; every object and subject in the system has labels (a set of tags represents a label) and capabilities based on what reference monitor allows or disallows certain information flow. In the DIFC model, applications can express their policies by defining these tags and their associated capabilities and attaching them to their process, threads, or data (i.e., files). In the next section, we further explain the DIFC model and abstractions in-depth, using the standard OS-based DIFC model in FlowROS.

Past works such as Asbetos [68] and HiStar [74] are examples of OS-based DIFC systems implemented entirely as a modified OS from scratch. They also use DIFC models such as labels/capabilities but are heavy as they change core OS abstractions to adhere to security and integrity requirements. HiStar uses page granularity and page protections to enforce information flow control policies. Labels are attached to every page in the system, and upon page access, the page protection mechanism triggers a reference monitor to identify and analyze the flow. Though it provides a security guarantee, the methodology is inefficient regarding execution time

and memory utilization (i.e., due to memory fragmentation). Also, it makes the enforcement heavy as the DIFC reference monitor requires memory management at the page granularity, leading to an increase in the execution time. On the contrary, flume implements its reference monitor entirely at the user level and tracks access of objects by subjects using the granularity of address space.

Architecture-level DIFC. Past works such as RIFLE [67] demonstrate an architecture-based runtime DIFC system capable of enforcing user-specified IFC policies for any program. RIFLE employs binary translation to convert ISA into Information Flow Secure (IFS) ISA [67, 60], and utilizes modified hardware to assist in tracking information flow. During runtime, the modified IFS ISA interacts with the trusted operating system, which is responsible for enforcing policies. Similarly, Loki [75] demonstrates that IFC policies provided by the application in DIFC can be further integrated into the architecture by leveraging the concept of tagged memory (i.e., tagging every word in physical memory) and assigning a 32-bit tag value per page in memory to monitor IFC flows at the processor level. This methodology can mitigate exploits due to a compromised OS kernel, but it still requires a minimized trusted kernel code to make final IFC decisions. It uses HiStar as a modified OS, which is not a commodity OS. In conclusion, architecture-level DIFC systems provide slightly better information flow control than mere OS-level enforcement, but ultimately they still rely on OS-level enforcement to make IFC decisions. Unlike OS-level and PL-level DIFC systems, they are not ad hoc in nature [60].

In FlowROS, we leverage the LSM architecture that is already present as a part of the Linux kernel to implement our reference monitor. Recent DIFC works, such as Weir [44] and Laminar [60], use the LSM framework for their DIFC model, as it can readily be used rather than requiring changing the whole OS.

2.3 Motivation

ROS is used on many platforms to accelerate application development for robotics since ROS2's communication middleware DDS provides seamless data transfer protocols and better management of devices and applications [19].

Unfortunately, ROS does not provide any security by default; any application can publish to any topic, and any application can subscribe to any topic. This leads to various attacks on ROS, such as spoofing messages by publishing on topics it is not authorized to, snooping on other applications' data by unauthorized subscription, or faking the application's identity [23, 16]. The ROS community came up with SROS [71], an added layer in ROS middleware, in an attempt to solve this problem. SROS makes it mandatory for applicants to have identities

backed by X.509 certificates, that a trusted third party signs and messages between applications are secured using the TLS protocol. Moreover, SROS mandates applications only to publish or subscribe based on the application manifests. The content of the manifest is also bound with the x.509 certificates and, therefore, cannot be forged by an attacker. The added SROS layer in ROS middleware prevents some basic attacks from happening, but it still does not completely prevent data leaks from happening in the ROS environment. For example, prior work [19, 16] shows shortcomings and attack scenarios even in an SROS secure layer on a ROS system. To highlight the ROS2 security, figure 2.3 shows the overall architecture for the DDS security [46]. The figure shows all the available security mechanisms, and these mechanism try to solve several attacks and loopholes, as shown below:

- **Authentication** This plugin identifies ROS applications running on the system or the users initiating the tasks on the DDS layer and also helps authenticate participants in the ROS environment.

- **Access Control** As ROS applications use the pub/sub model to send and receive data through DDS middleware, this service plugin enables the authorization of the operations. For instance, is it allowed to join a domain and publish or subscribe to specific topics, etc.?

- **Cryptographic** plugins enable the DDS layer to provide operations to secure communications at the transport layer so that an eavesdropping application cannot leak the information, which includes data-level encryption, decryption, hashing, etc.

- **Logging and tagging** plugins. Logging plugins provide application interfaces for auditing ROS-level DDS events. Moreover, tagging allows ROS participants to tag their messages at the DataWriter level. Secure DDS [46] provides applications the capability to attach tags to their published messages, and the access control plugin of DDS will use these tags to authorize the publish/ subscribe operations performed by other participant nodes in the system or communication graph. Note that applications must provide access control rules in an XML file describing the rules. The tagging plugin gives applications fine-grained control over their data, specifying who can subscribe to certain data classes. However, it is only enforced at the DDS layer, meaning a malicious application can still send or receive data using OS-level communications.

ROS applications can use these plugins to prevent several attacks at the ROS2 middleware layer [16, 23, 24, 19]. However, these mechanisms still do not provide end-to-end security guarantees due to the following reasons:

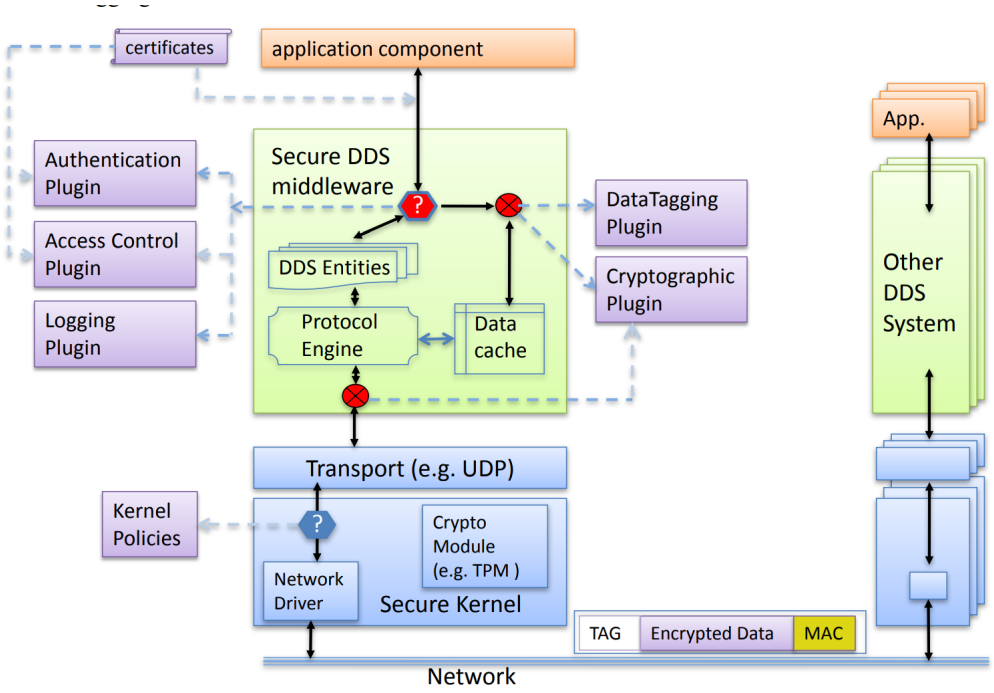


Figure 2.3: DDS security mechanisms as an added plugin (Image borrowed from OMG DDS documentation [46])

1. **Lack of end-to-end reasoning**:
   SROS strictly adheres to the application manifests and only allows applications to publish or subscribe to topics declared in their manifest. Application writers write these manifests, and they cannot know a priori what other applications will be executing on the platform and, hence, will be writing manifests in a manner such that the application can work in all possible runtime environments.

2. **Bypassing the ROS layer completely**:
   A malicious application that wants to ex-filtrate sensitive information can create a socket, file, pipe, etc., without using any API service from the middleware and communicate the data using these OS-level abstractions by completely bypassing the ROS layer.

The past work [16] demonstrates the efforts to mitigate such loopholes and vulnerabilities by introducing an added layer of trusted software, a full-fledged system-wide mandatory access control (MAC) enforced by the underlying operating system. Privaros applies host policies on the ROS applications running on the drone using kernel-level enforcement (AppArmor). It requires a trusted system admin to identify centralized security requirements and prepare policies to be enforced by kernel-level enforcement to prevent malicious applications from leaking

users' sensitive information to the external untrusted sinks. On the contrary, FlowROS provides a decentralized information flow control for robotic applications targeted specifically for ROS2 middleware, as it is widely used for developing robotic applications. FlowROS enables ROS applications to express information flow policies based on how their sensitive data flows in the system and control its propagation.

2.3.1 Threat Model

In practice, ROS applications use APIs exposed by ROS middleware to create publishers/subscribers and clients/servers for communication at the application level. Further, ROS middleware implementation uses APIs exposed by the DDS layer. The DDS layer creates files, sockets, shared memory, etc., for communication between applications, providing an abstraction layer to ROS. DDS encapsulates its implementation within its libraries and modules, preventing applications from inadvertently modifying or overwriting the internal DDS implementation. Our threat model includes a malicious application that can create files, sockets, or shared memory using OS-level abstractions and ex-filtrate sensitive information by completely bypassing ROS, SROS, and DDS layers. Therefore, we require OS-level information flow control to intercept all the sensitive communication in the system to prevent sensitive data from leaking. FlowROS's Trusted Computing Base (TCB) includes an OS-level enforcement engine and DDS library. We require DDS library support to distinguish meta traffic that is only being used by the DDS layer from actual data transfer. We modified the DDS layer to communicate metadata traffic-related ports to the kernel to avoid inconsistent information flow leading to label explosion. Note that meta-traffic ports are not available to applications and are only used by the DDS layer. Since the DDS layer is part of our TCB, we do not mitigate the attacks that can overwrite or modify the DDS binaries linked to ROS applications. We also do not consider the covert-channel attacks that are possible due to the implicit label propagation feature for FlowROS for unmodified applications, as it can be prevented by allowing only explicit label propagation, but that would require all the applications running on a ROS-based system to be modified.

2.4 Challenges

2.4.1 Label Explosion

In a DIFC system, labels represent some security class the objects and subjects belong to. The reference monitor performs information flow control checks based on these labels. Explicit

label propagation is when a process or a subject explicitly changes its label. Similarly, implicit label propagation is when a reference monitor changes the label of a process or subject based on an information flow it encountered in the system. Note that implicit label changes only happen upon valid communications (i.e., the receiving process can be allowed to receive that information). Consider ROS applications running in a ROS-based environment shown in figure 2.4. ROS applications are linked with Data Distribution Service (DDS) to communicate. The Endpoint Discovery Protocol (EDP), part of DDS, establishes communication between ROS applications by broadcasting meta-traffic messages to all applications running on the system. Consider an application P publishing some data on a topic that Q has subscribed to, leading to an information flow from P to Q. A communication from P to Q would change Q's label to P's label, but due to the broadcasting nature of DDS's discovery protocol, which is responsible for endpoint discovery of applications trying to communicate based on the Pub/Sub model. The DDS layer internally sends packets to all the applications running on the system for endpoint discovery protocol. Discovery leads to an information flow that was not an actual communication but rather EDP meta-traffic that implicitly raises R and S's security label equal to P's label, even though R and S have still not seen the data associated with security class L_P. Meta-traffic broadcasting creates implicit label propagation, leading to label creep, which unnecessarily restricts R and S. Note that the implicit label propagation only happens when R and S are indeed allowed to subscribe to the information published by P (i.e., the reference monitor will perform implicit label propagation if capabilities of R and S allow it to do so).

DDS uses multi-cast ports for its endpoint discovery protocol. These are some well-defined ports that the discovery protocol uses at the DDS layer only. In FlowROS, we register these ports in the kernel through the DDS layer by modifying EDP to avoid information flows leading to label explosion. Note that ROS2 applications can configure the listening locators through XML configuration files, but DDS does not expose the ports used for discovery protocol to applications [6, 10].

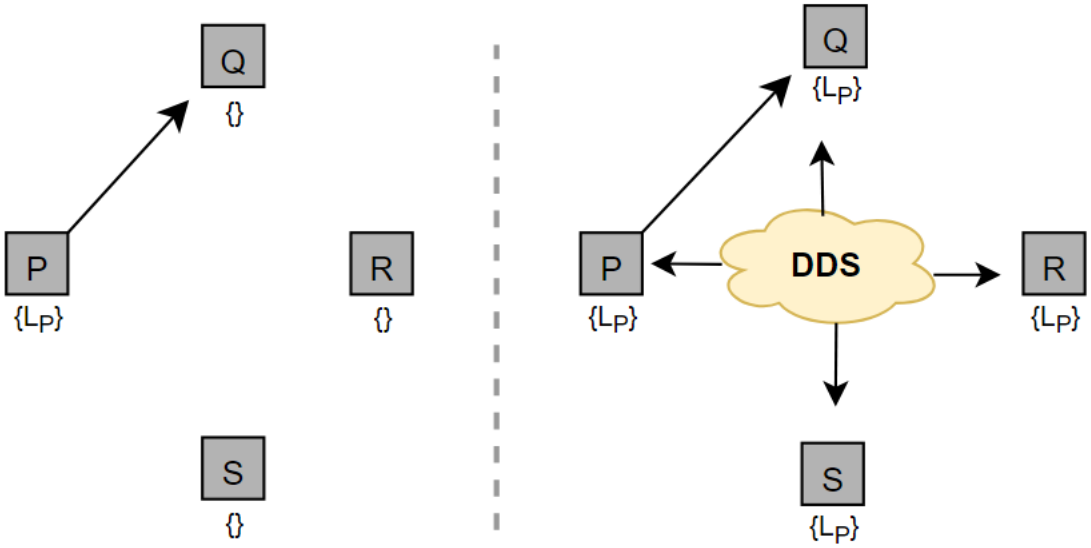


Figure 2.4: Label explosion due to discovery EDP protocol

Chapter 3

Related Work

We demonstrate the use of the DIFC system in ROS-based environments applicable on many robotics platforms where ROS applications can leverage a decentralized model for controlling and securing their sensitive data from accidental disclosures and preventing malicious applications from gaining access to it. Further, this chapter aims to explore the works coinciding with ROS and existing DIFC systems that attempt to solve similar security and privacy-related concerns.

3.1 ROS Security

ROS, in its basic form, lacks certain fundamental security mechanisms leading to various kinds of attacks that are possible. Mclean et al. [39] highlight an attack that is possible on a ROS-based system; a ROS participant is able to inject false commands into an actual robot running in a ROS environment, as ROS messages are by default not encrypted therefore making it possible for an attacker to capture ongoing messages and reuse them to generate false commands. Similar works highlight the shortcomings of ROS [54, 29] leading to various kinds of attacks, such as denial of service, privilege escalation, spoofing, etc. ROS community developed a few mechanisms as an added library support to mitigate these kinds of attacks SROS [71]. SROS provides features such as TLS support for socket-level ROS communications and x.509 certification support to restrict the computing nodes to well-defined namespaces and permitted roles. It also provides tools for ROS applications to generate access control policies using AppArmor profiles. Further, it confines the nodes executing in the context of a process at the OS level. Hardware Robot Operating System (H-ROS) [36] provides infrastructure for robot vendors to use the H-ROS tools built upon ROS to create their robotic components that support real-time functionalities, better performance, and security. It protects robotic components due to its hardware-supported encryption and authentication mechanisms.

TROS [37] presents an attack scenario on humanoid social robots where an attacker can land on the system software stack, most probably through network interfaces, and try to misbehave with its functioning resulting in a susceptible situation. TROS mitigates such attacks on the system by introducing a Hardware-base Trusted Execution Environment (HTEE) and segregating critical functionalities of ROS (i.e., keys, IDs of nodes, URI of ROS entities, etc.) to a trusted execution environment using Intel SGX and separating the memory that might consist of sensitive information and leveraging attestation for some functions as well.
There are works that [14, 51] demonstrate how publish/subscribe-based systems can be secured by integrating with a Role-based Access Control (RBAC) [25]. RBAC is suitable for large-scale systems targeted for an organization or enterprise-based domains (e.g., public bodies) where the roles of entities follow a straightforward hierarchical structure, benefiting the RBAC-based model. The idea behind RBAC systems is to disintegrate the relationship between principals and their privileges and assign roles (or rights) to services that principals will be using. It makes it simpler and applicable for large-scale systems as the number of individual users is very high and keeps changing.
Recent work by Hwimin Kim et al. [31] shows how the security of DDS can be improved by adding an attribute-based access control model to verify and authorize communication between participants. They show ABAC is naturally suited for DDS as DDS entities are well associated with their attributes. And DDS performs access control by allowing/denying the connection requests at the DDS discovery phase itself.

Beck et al. [16] demonstrate the shortcomings of the added SROS layer in ROS wherein a malicious application can directly use OS-level artifacts in order to communicate or exfiltrate sensitive information to untrusted sinks in the system. To mitigate such loopholes, they leverage LSM-based AppArmor to restrict the information flow in the system by using the AppArmor application profiles given by a trusted centralized authority. MAC systems can prevent sensitive data from leaving the system but with the condition that it requires complete system-wide policies such that it covers all possible communication paths in the system [3, 65]. The coarse granularity of MAC policies puts undue restrictions on data that does not belong to a sensitive class [60]; also, generating a whole system-wide policy is difficult to achieve, which makes it not very application-friendly [59]. Considering these drawbacks of a mandatory access control system, we leverage a DIFC-based model for ROS applications to better control their sensitive information, and we demonstrate a use-case of DIFC on robotics platforms.

Dieber et al. [23, 24] shows how fault injection of messages and similar attacks can be prevented by adding a new mechanism as an Authentication server (AS) on top of ROS without making any changes to the underlying ROS layer and verifying the inputs from nodes and/or components in the system using cryptographic methods. There are works [54, 55] that identify the fingerprints of malicious activities using automated tools and then prevent it from happening by generating rules automatically and creating a firewall.

3.2 Information Flow Control

MAC systems [65, 3] can very well enforce information flow control (i.e., secrecy and integrity rules [22]) by labeling subjects and objects in the system. These labels are static in nature and require external interference for transitions. For example, once a policy is loaded into the system, it cannot change its label without requiring to reload the policy itself. As discussed earlier, MAC systems suffer coarse granularity; as a result, work such as SEApp [59] in this domain enables application developers to control and confine (or sandbox) different modules of applications with a clear requirement to protect the system resources from malicious modules or applications. SEApp brings a programmable interface to application developers so that they can use SEAPP-exposed APIs to label files, Android services, and components. And use SEApp policy language to define access rules with respect to labels. Note that SEApp uses SELinux as their enforcement mechanism, which is a MAC-based model.

A decentralized information flow control [68, 33, 74, 60, 32] system naturally provides this capability to applications, such that they can confine other components of their applications based on the sensitive data the components are dealing with and also segregate the different class of sensitive information within single process context by dynamically changing their labels.

Browserflow [48] demonstrates a taint tracking system for cloud-based document apps to control the flow of "texts" across cloud services and prevent accidental data disclosure. They highlight cases where certain sensitive texts may belong to users or enterprises, and dissemination of those texts by an attacker to docs in other browser tabs on the same host or across remote hosts may violate the user's privacy policy. They prevent text disclosures based on user-given policies by enforcing information flow control in the browser. The novelty in their approach is how they tag text segments inside documents and monitor the propagation of those texts across other docs if a user tries to copy/paste it or if a malicious application attempts to exfiltrate it out of the system.

Historically, Android leverages DAC and MAC-based access control mechanisms [63, 65, 59] to protect the application's private and user data from accidental data disclosure. A natural need for bringing control to application developers led to decentralized information flow control

on Android [43, 30, 73], using which developers can have fine-grained control over the propagation of their sensitive information. Weir [44] highlighted some drawbacks of the DIFC work on Android. Some of these works support floating labels for background components to give backward compatibility as they are unmodified and used by all applications, and that creates a label creep as labels will be merged when more than one application is creating background components, leading to components not being usable anymore because of the undue restrictions. Aquifer [43] does not support labeling the background components initialization to support multitasking for all the applications that use it; therefore, it does not prevent data disclosure from these components. Weir presents a DIFC system practical for Android's unpredictable information flow by creating fresh instances upon every new intent with a different security context, namely "polyinstantiation." All the DIFC systems mentioned earlier implement process-level reference monitors, assuming only process-level components. Note that the Android-based environment has unpredictable flow due to user interference. In contrast, a ROS-based environment generally does not suffer unpredictable flows; therefore, indicating FlowROS is practical for such an environment as application developers are fully aware of communications among ROS nodes that are part of the application context. FlowROS is a process-level DIFC system capable of enforcing information flow control policies at the process level. Additionally, FlowROS makes DIFC feasible for platforms like ROS by addressing the issue of "label explosion," a common challenge for any DIFC system.

Chapter 4

FlowROS

In this chapter, we explain the DIFC model that FlowROS use, including the terms tags, labels, capabilities, and label propagation. Then we present the implementation of FlowROS using the LSM framework and highlight changes we added in the LSM for ROS applications. Lastly, we demonstrate some DIFC policies for ROS applications.

4.1 DIFC Model

We define the DIFC abstractions used in FlowROS. FlowROS's DIFC model is quite similar to the previous DIFC models [33, 44, 60, 68], and we highlight the changes in detail further in the section. In the DIFC system, principals are the entities (i.e., users [33], processes [42], and kernel threads [74]) that perform operations on files or data in the system.

4.1.1 Tags and Labels

The fundamental terms for a Decentralized Information Flow Control (DIFC) system are tags, labels, and capabilities. **Tags** are arbitrary tokens chosen from a large set of opaque tokens (denoted as $\mathcal{T}$). Tags are utilized to associate data with a particular security class or level. The tags themselves have no inherent meaning; it is the application that imbues the tag with semantic meaning by associating it with its data (e.g., an application might use arbitrary tags t_1 and t_2 for data belonging to two different security classes). For example, application A might attach tag `a` to its sensitive data, while application B might use tag `b` for its sensitive data. A principal (i.e., a process) in the system can define one or more tags to fulfill the secrecy or integrity requirements of its sensitive information.

A **label** is a set of tags drawn from $\mathcal{T}$, as shown below. In other words, a label can be formed from a collection of tags or a single tag. Note that a label can also be an empty set (i.e., containing no tags). Labels serve as the basis for any Information Flow Control (IFC) reference monitor to make information flow decisions;

$$L = \{t_1, t_2, \ldots, t_n\}$$

4.1.1.1 Partial Order Lattice

In the partial order, a Label L_x is considered less than or equal to a Label L_y (i.e., $L_x \subseteq L_y$) if all the tags that are there in L_x are also there in L_y. Hence, in DIFC, the partial order often implies a subset relation between labels, and this subset relation dictates how data flows in the system; that is, information can flow from a lower label to a higher label but not vice-versa. Applications are required to assign appropriate labels to their processes based on the security requirement; therefore, after assigning labels in adherence to the security policy, all the labels in the system follow a partial order lattice relation [62, 44]

4.1.1.2 Classification and Declassification

Since the advantage of DIFC over MAC systems is that it allows label changes at runtime, the operations enabling this are known as classification and declassification. Classification involves assigning labels to data or a process. This label indicates the data's security class. Declassifying a label means decreasing its security level by removing one or more tags from it, thereby allowing data that was once restricted to be processed at a lower security level. Declassification is very powerful in terms of flexibility and functionality, as it facilitates the downward flow of information to lower security levels by declassifying information that was derived or accumulated from various sources with different security classes.

All the principals (i.e., processes) in the system have their own set of labels. The reference monitor in the DIFC systems use these labels to mediates the information flows that are happening in the system (i.e., per process L_S for secrecy and L_I for integrity.). The DIFC system uses a secrecy label to allow/disallow access to sensitive information and avoid accidental data disclosure to unidentified principals and untrusted sinks and Integrity labels to protect the data from corruption. For instance, consider a process P with tag $\mathtt{t} \in L_S$, and then the reference monitor conservatively assumes that Process P has seen the data tagged with t. For it to release or sent to any lower-class label, it requires declassification. Similarly, for integrity, if $\mathtt{t} \in L_I$, process P performing some computation using input files or sockets should have an integrity level equal or super-set of L_I to maintain the integrity of process P. In theory, secrecy, and integrity labels can have any tags, but in practice, they are mutually exclusive.

As discussed, secrecy and integrity labels are used to enforce and implement secrecy and integrity policies. Every principal in the system initially has an empty label which it may use to configure policies for its data; these labels form a partial order lattice under the subset relation, and the bottom of this lattice represents unlabeled resources. Objects in the system also have an empty label initially. If any labeled process creates a file or an object, then those object inherits those labels. Later access to these files will be allowed if it complies with the DIFC label propagation rules [33, 44].

Now the question arises of how the reference monitor knows who has the privilege to add or remove tags from their labels (i.e., classify and declassify). The answer is capabilities; every principal in the system has a capability set, C_p, that represents whether a principal (i.e., process) can raise its label to a higher secrecy level by adding a tag or decreasing by removing a tag. For instance, let us consider a tag t, such that $t^+ \in C_p$: a t^+ gives the principal the privilege to add a tag t to its label. Adding tag t to its label allows the process to release the data or consume the data corresponding to the tag t of the security class. Similarly, a principal that creates a tag t owns the data for security class t and is capable of giving t^- capability to other principals in the system; As a result for $t^- \in C_p$, a t^- gives the principal the privilege to remove a tag t from its label (i.e., declassify) [33, 68, 44]; removing t will signify a declassification, and then the principal can communicate to another principal or access a file not having tag t in its label. In a centralized IFC system (i.e., MAC), the creation of the policies can only be done by a trusted "system admin" who can create new tags based on the system requirement and can add or remove tags from secrecy labels (classify or declassify information), or add or remove tags to integrity labels (endorse or drop endorsement). In FlowROS, similar to Flume [33], Asbestos [68], and Laminar [60], individual principals can create new tags. The principal P who creates a tag t, by default, has t^+ and t^- capability (i.e., if P creates tag t, then $\{(t^+, t^-) \in C_p\}$). Moreover, P can further give t^+ or t^- to any other principal in the system.

4.1.1.3 Global Capability

Similar to a per-process capability, the concept of global capability G_p is useful when a principal wants to give everyone else in the system the capability to raise their label or to downgrade their label as the data owner (i.e., principal), who create new tags for its data cannot individually give each process t^+ or t^- capability, and also no process or principal can know priori what all processes are executing in the system with what process ID to do so.

4.1.2 Label propagation

In DIFC, applications give the policy to control the flow of its data (i.e., propagation) and access its files using labels. It attaches tags to the processes and objects that it wants to control. Let us consider an information flow from x to y, where x can be a principal trying to access an object or a file y to write, or y can be another principal that principal x is trying to communicate to send some data, in that case, information is flowing from x to y. Similarly, x may be trying to read from a file y, or x may be trying to receive some information from y; in that case, we say information flows from y to x. To define formally, FlowROS enforces the DIFC rules for data flowing from x to y:

4.1.2.1 Secrecy

In the partial order lattice relation of the labels, Bell and LaPadula [62] define a secrecy policy as when information is flowing from x to y, a principal is not allowed to read data from the higher level (i.e., no read-up) and is not allowed to write any data to the lower level (no write down). Formally, S_x and S_y are the secrecy label for principals x and y, respectively; secrecy is preserved if and only if the information flow from x to y follows:

$$S_x \subseteq S_y$$

To comply with the above information flow secrecy rule, the principals may explicitly change their label by adding or removing a tag from their label. For example, to satisfy the secrecy rule, principal x can add a tag t to its label S_x (classify) if and only if it has the capability to do so (i.e., $t^+ \in C_x \cup G$), where G is the global capability. Similarly, y can remove a tag t from its label S_y if it has the capability to declassify tag t (i.e., $t^- \in (C_y \cup G)$).

4.1.2.2 Integrity

The integrity rule defines the alteration rules to maintain the integrity of the processes or principals performing computation. It does not allow reading from lower levels (no read down) and no writes to higher integrity levels (no write up). To formalize the enforcement rule [20]:

$$I_y \subseteq I_x$$

Similar to the secrecy rules, any principal x can satisfy the integrity rule by endorsing its label I_x to send the information to a higher level, if and only if x has the capability to do that in C_x and while receiving the information from lower level labeled principal x, y can drop the endorsement by removing the tag, let say t, again if and only if it has the capability to remove tag t from is label I_y (i.e., $t^- \in (C_y \cup G)$).

4.1.2.3 Label Changes

DIFC systems enable programs (i.e., at the process level) to implement policies. As a result, programs have the capabilities to control and use the APIs and system calls exposed by the DIFC enforcement engine to create tags, classify and declassify information, endorse or drop endorsement for the safe information flow and access control [74, 68, 33, 44, 60]. As discussed earlier, to satisfy the secrecy or integrity rules, a principal can add or drop its label using its capabilities requires the principal to **explicitly** change its label. In FlowROS, if a principal 'P' wants to raise its label from L_1 to L_2, by adding the tags to L_1 that are not there in L_1 but present in L_2, it should satisfy the following equation:

$$(L_2 - L_1) \subseteq (C_p^+ \cup G^+)$$

Similarly, P wants to drop endorsement of its label L_1 to L_2 by dropping some tags that are there in L_1 but not present in label L_2; it should satisfy:

$$(L_1 - L_2) \subseteq (C_p^- \cup G^-)$$

Explicit labels

Tranquility [62, 44] is the final assignment of labels to objects and principals in the system; it is the property of the MAC systems. However, this property is relaxed in DIFC systems, so the principals may change their labels (classify or declassify) "safely." Therefore, changing the immutable labels in MAC to mutable labels in DIFC makes it more flexible for applications. But it also can't dwell with the unpredictable information flow environments since all the applications in the system need to be modified to be compatible with DIFC. Explicit label propagation requires principals to explicitly change their labels using programming (i.e., modify applications to call label change mechanisms); it includes creating tags for their sensitive information and adding/removing tags from their labels, declaring capabilities to control the flow of their sensitive information. Listing 4.1 shows a ROS2 application program that creates the tag for the data it wants to control. It always increments or raises its label by adding that tag before publishing the sensitive information. Figure 4.1 shows how FlowROS's reference monitor makes the information flow decision using labels. In the case of modified applications, principals must always change their labels to satisfy secrecy and integrity rules. A flow from principal 'A' to principal 'B' will result in a complete denial due to label mismatch.

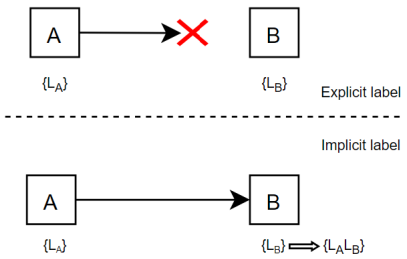


Figure 4.1: Implicit vs. Explicit label propagation

Implicit labels

The concept of floating or implicit labels emerged to provide compatibility for unmodified applications. Asbestos used it [68] and subsequent works as well [43, 30, 44]. It was introduced to fill the compatibility gap between modified and unmodified applications, making DIFC systems usable in a practical environment (i.e., where information flow is unpredictable). Implicit label propagation allows the information to flow from a principal x to y, such that the resulting L_y will equal $(L_y \cup L_x)$, where y is a process belonging to an unmodified application. X can be a process that belongs to a modified or unmodified application [44, 68]. Figure 4.1 shows how the reference monitor combines the label L_A and L_B due to an implicit flow of information. However, note that kernel only performs this merging of labels for unmodified applications if and only if their capabilities allow it to do so.

4.1.3 Implementation

FlowROS consists of mainly three components 4.2, user IOCTL interface for ROS applications, kernel IOCTL interface, and OS-level reference monitor. We implement the FlowROS DIFC system on Linux kernel version 4.9 (i.e., vanilla). FlowROS's reference monitor is implemented from scratch using Linux security modules (i.e., LSM[72]). MAC systems (e.g., SELinux, AppArmor, Smack, etc.) also use the LSM framework to implement kernel-level enforcement to apply centralized policies. We further explain in this section a brief on FlowROS's reference monitor.

4.1.3.1 Why LSM?

A basic LSM framework provides already compiled code directly inside the Linux kernel, namely hooks, which allows us to insert checks into the kernel through hooks that are invoked whenever something sensitive is going to happen in the system. For instance, a sensitive operation means that when a process tries to access files, credentials, sockets, and inter-process communications, the hooks will control security modules to perform enforcement.

4.1.3.2 Major and Minor LSMs

The major LSMs can load/unload files from the user space at run-time and when the system boots. It also gives exclusive access to pointers and security identifiers in the kernel objects for security modules. SELinux, AppArmor, TOMOYO, and SMACK are examples of major LSMs. At a time, only one LSM can be loaded into the system because the LSM framework provides exclusive access to security fields already embedded in kernel objects. On the contrary, minor LSMs require fewer security fields and security context pointers. Hence it is loaded along with a major LSM in a stacked manner. Minor LSMs only need flags to turn specific fields on/off. Examples include YAMA, Lockdown, etc. We implement FlowROS's reference monitor as Major LSM since we require security context pointers and identifiers present in the kernel objects to create and initialize DIFC fields in every kernel object and subject (i.e., process) and also to mediate all the information flow in the system; as a result, reference monitor can allow or disallow information flow (e.g., using label propagation rules).

Below we show what structures we use in the kernel to initialize security contexts for objects and principals. Each process in the system has set of Labels and a capability set.

- Process
  `struct task_struct and struct cred` Since Linux uses a lightweight process architecture, it keeps `task_struct` for every thread in the execution phase. FlowROS is a process-level enforcement, not a thread-level. Thread-level enforcement requires separating the memory that thread can access [60]. Therefore, we initialize security context (i.e., labels) in `struct cred`, which is a process-level structure. Note that we do not support thread-level DIFC, but we mediate all the accesses in the systems. Suppose there is more than one thread executing in a single process context. In that case, other threads will be blocked while FlowROS performing label checks on one of the threads (i.e., process-level enforcement limitations [33, 43, 44]).

- File and sockets.

`struct inode and struct file` We label `inode` and `file` so that the reference monitor can keep track of its access.

- Packets.
  `struct sk_buff` In Linux, packets are represented as `sk_buff` Since DDS communications are mostly network-based, we store the sender process's PID in the secmark field of `sk_buff`

LSM framework provides security pointers (i.e., `*security, *i_security, *f_security, etc.`) that we can use to store the security context for the reference monitor. As a result, all the objects in the systems initially have empty labels, and objects inherit the labels from the principal upon first access. Furthermore, label changes later follow security rules (secrecy and integrity). FlowROS uses `*task_security, and *cred_security` for process-level security context.
Note that the LSM framework provides more support to mediate all kinds of information flow in the system, which SELinux, AppArmor, and similar standard major LSMs use. For example, `struct new_device` for controlling the network devices, `struct kern_ipc_perm and struct msg_msg` for System V IPC protocols. Nevertheless, as we implemented FlowROS as a prototype, we limited the implementation to a proof of concept. FlowROS only registers and defines access check hooks to cover basic decentralized information flow control policies and mitigate accidental data disclosures at the network and file system level (i.e., we do not mediate the flow of shared memory and IPC). Implying a malicious application can leak sensitive data through IPC or shared memory communication that FlowROS currently does not monitor. However, that is not a vulnerability or a bug; it is instead a prototype limitation. It can easily give a complete OS-level end-to-end security guarantee in the future. In FlowROS, we register the following hooks that are important for initializing and allocating security context for the reference monitor for all objects and principals in the system. All principals in the system initially inherit the empty labels and capabilities from the init process. However, later, these labels and capabilities can be used to define their security policies. FlowROS currently exempts the init process and system-level services (i.e., daemons) from LSM security checks. However, FlowROS also supports LSM infrastructure to modify their labels and capabilities because FlowROS allocates empty labels and capability set at the boot-time for them.

- File related hooks
  `file_open, inode_create,inode_alloc_security,`
  `inode_free_security,sb_alloc_security,sb_free_security`
  `file_ioctl, file_receive`

- Socket related hooks
  `sk_alloc_security, sk_free_security, socket_sendmsg, socket_recv_msg,`
  `socket_sock_rcv_skb`

- Task and Cred relate hooks
  `task_alloc, task_free, cred_alloc_blank,`
  `cred_free, cred_prepare, cred_tranfer`

4.1.3.3 Limitations

Note that the LSM framework provides more support to mediate all kinds of information flow in the system, which SELinux, AppArmor, and similar standard major LSMs use. For example, `struct new_device` for controlling the network devices, `struct kern_ipc_perm and struct msg_msg` for System V IPC protocols. Nevertheless, as we implemented FlowROS as a prototype, we limited the implementation to a proof of concept. FlowROS only registers and defines access check hooks to cover basic decentralized information flow control policies and mitigate accidental data disclosures at the network and file system level (i.e., we do not mediate the flow of shared memory and IPC). This implies that a malicious application can leak sensitive data through IPC or shared memory communication that FlowROS currently does not monitor. However, that is not a vulnerability or a bug; it is instead a prototype limitation. It can easily give a complete OS-level end-to-end security guarantee in the future.

4.1.3.4 DDS Modifications

ROS2 is a new version of ROS developed using a Data Distribution Service (DDS) that is responsible for managing the data exchange among ROS applications using the wired protocol. FlowROS requires support from the DDS layer, which is linked to all the ROS2 applications in the system. DDS internally manages many layers that separate different modules responsible for efficiently managing data transfer over a network or other mediums that DDS supports (2). Mainly, two important protocols send data over the network using sockets or file OS abstractions. First is the discovery protocol, which helps ROS applications discover each other and introduce their publish/subscribe ports for later data transfer. Moreover, the second is the

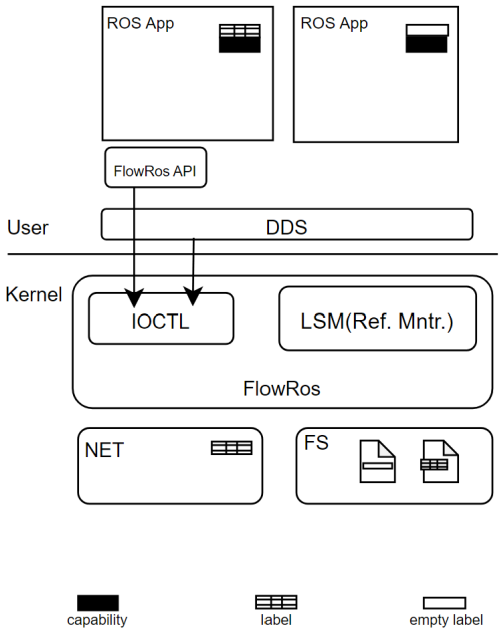


Figure 4.2: FlowROS architecture that includes mainly three components: 1. User API for ROS applications to manage labels that communicates through IOCTL interface, 2. DDS middleware support to register ports used in Endpoint Discovery Protocol (EDP) and 3. Reference monitor that is part of LSM

protocol that the publish/subscribe model of DDS uses to send the actual data produced and consumed by the ROS applications. The DDS layer only uses the discovery protocol, and its data is not exposed to the upper layers. As we highlighted, the discovery protocol leads to a classic "Label Explosion" problem, making DIFC or any taint tracking system impractical to use at the level, as the reference monitor will deny all the subsequent accesses in the system due to the label creep. Therefore, FlowROS figure 4.2 modifies DDS layers to segregate the ports used for the discovery protocol only by the DDS layer by registering these ports into the kernel. The reference monitor does not enforce IFC rules on discovery, hence adding the DDS layer to the trusted computing base for the FlowROS DIFC system.

4.1.3.5 Domain Declassification

FlowROS allows domain declassification [15, 66, 44], where applications that create tags, as data owners, can declare a set of trusted network domains (t^D) for the security class t.As a result, FlowROS will declassify the information with the tag t being exported to the network. FlowROS allows the DDS protocol to send packets to the local host (i.e., 127.0.0.1) by default, as DDS communicates between processes using the local host IP address. However, to enforce information flow control policies, FlowROS applies secrecy and integrity rules at the network `sendmsg` hooks as well as `recvmsg` hooks, and upon any violation, it can drop the packets at the network interface itself. Since DDS mainly uses network-based communications using packets, both for discovery protocol and data transfer protocol. It makes it challenging for the reference monitor to get the receiver's security context at network LSM hooks and the sender's security context at the receiving end. Therefore, FlowROS's reference monitor embeds the pid of the sending Process in the already existing security field of packets (i.e., secmark) at the LSM hook level and uses that security field to monitor DDS communications to apply information flow control policies at the receiving end. (e.g., Privaros [16] also uses this method at the LSM level). Also, ROS2 middleware is used for applications running in a distributed setup.

4.1.4 User APIs

`create_tag(tag)`

Creates a new tag with the specified name.

`add_tag_label(tag)`

Adds a tag to the label of the calling process, allowing it to be associated with principals and objects.

`remove_tag(tag)`

Removes the 'tag' from the label of the process.

`add_global(tag)`

Function to add a global tag, allowing any process in the system to consume the data that belongs to the global tag.

`add_pos_cap(tag, pid)`

Function to give process privilege to increment its label with the given 'tag.'

`add_neg_cap(tag, pid)`

Function to give process privilege to decrement its label with the given 'tag.'

`add_domain(tag, IP)`

Allows applications to declare domain declassification based on the specified IP address.

4.2 Sample DIFC Policies

4.2.1 Taint Tracking

We will now explain how a ROS2 application can implement a simple taint tracking policy for its sensitive data to avoid accidental data disclosure to the untrusted sinks. In figure 4.3, consider a process P^m that belongs to a ROS application; superscript m represents that to apply a taint tracking policy, process P^m is modified at the program level for changing its label L_p by adding the tag t before sharing the sensitive information tagged with t. As P^m creates tag t, by default, it has the positive and negative capability for tag t (i.e., $(t^+, t^-) \in C_p$). Now, P^m adds the positive capability t^+ to G_c so that any application can consume the sensitive data tagged with t, but only by adding that tag to its label. As a result, Q consumes the data from P^m with implicit label change by the reference monitor (i.e., $L_Q = \{\} \rightarrow \{t\}$). Note that the implicit label change for process Q is because it is an unmodified application; a DIFC system supports such

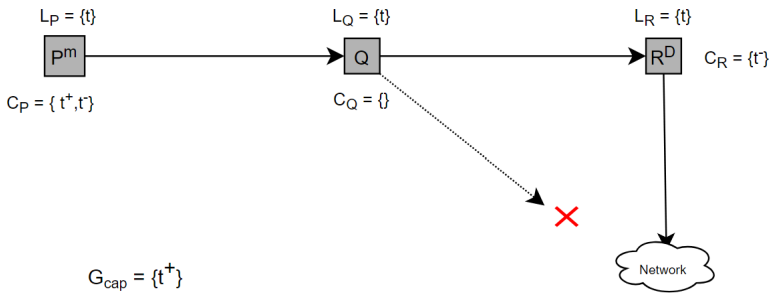


Figure 4.3: Simple taint tracking

implicit label changes to provide backward compatibility to unmodified applications[33, 44]. Q consumes the sensitive data produced by the P^m, but it cannot ex-filtrate sensitive information to untrusted sinks as it does not have declassification rights. Similarly, any other applications in the system will be able to read or consume the information tagged with t but will not be able to leak the data. Since P^m is the data owner for tag t, it specifies trusted declassifier R^D by giving declassifying capability to R^D (i.e., $(t^- \in C_R)$).Therefore, the reference monitor will allow R^D to send the sensitive data over the network by implicit declassification. Since R^D is a trusted declassifier, it can also explicitly change its label by dropping the tag t (i.e., $L_R = \{t\} \rightarrow \{\}$) and then sending it over the network.

4.2.2 Storage Policy

FlowROS supports labeling in the file system so that the applications can secure their data and avoid data leaks to untrusted applications or sinks. Figure 4.4 shows a storage policy where P has some sensitive information, and it creates a tag t for that information. Let us say P makes a file to write the data belonging to security tag t to the file. The reference monitor will initialize the file with the label of the principal that creates it (i.e., t). Therefore, for any application (e.g., Process Q in figure 4.4) with a label lower than the file's label L_f, the reference monitor will deny the access (i.e., restrict information flow if $(L_f \subsetneq L_Q)$). However, since process P has given classifying rights to the Process, R. R can access the file by explicitly modifying its label, or if unmodified, the reference monitor will change its label using implicit label propagation (following secrecy and integrity rules). Although process P, the data owner of t, has given t^+ capability to process R, R cannot ex-filtrate or leak the data after reading

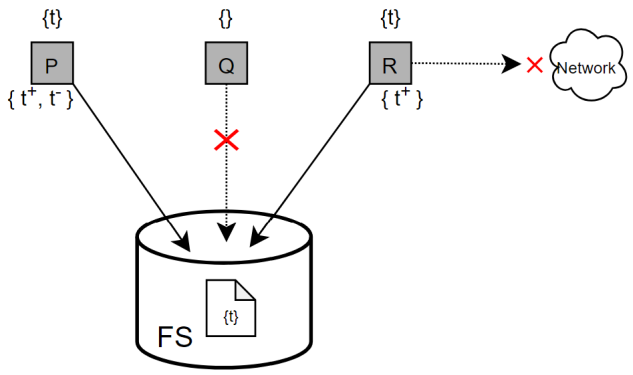


Figure 4.4: Storage policy

from a file with P's sensitive information. If R tries to send data over the network, FlowROS's reference monitor will not allow that, as R has a classified label with the security class t. Note that all the untrusted sinks in the system are, by default, considered to have an empty label, implying no data associated with any security class should go out of the system.

Chapter 5

Evaluation

In this Chapter, we first demonstrate a DIFC-capable modified version of the Camera application that can protect its data from accidental data disclosure. Then we show our experimental results highlighting overheads incurred due to DIFC modifications

5.0.1 ROS2 Camera Application

To highlight the motivation behind a DIFC system over a MAC system in terms of better usability, we identified an already existing ROS2 demo `image_tools` application [58] and modified applications to support DIFC and prevent disclosure of sensitive data, listing 5.1 shows a snippet of the application that publishes a live camera feed on the topic ImageRaw and publishes the status of the camera on the topic ImageStatus continuously while it is running on the system. Previous work [16] highlights a use case for applying access control policies in the robotics environment, where a centralized access control policy tries to mitigate data leaks and illegal access to sensitive information. For instance, a camera feed is considered to be sensitive information. Ex-filtration of it to untrusted sinks (e.g., network or storage) may cause a serious privacy issue. Still, other applications can consume it; for example, an image processing application might need that data to identify and avoid obstacles that might be coming in the way of a drone traveling on the fly. On the contrary, if the image processing application is executing on some cloud server to process the camera feed, the system must have a trusted declassifier to sanitize the video feed and send it over the network. Considering similar access control policies, we demonstrate the use of FlowROS to rectify the drawbacks of access control policies and the use of decentralized information flow control on the robotics platform that is more application-friendly.

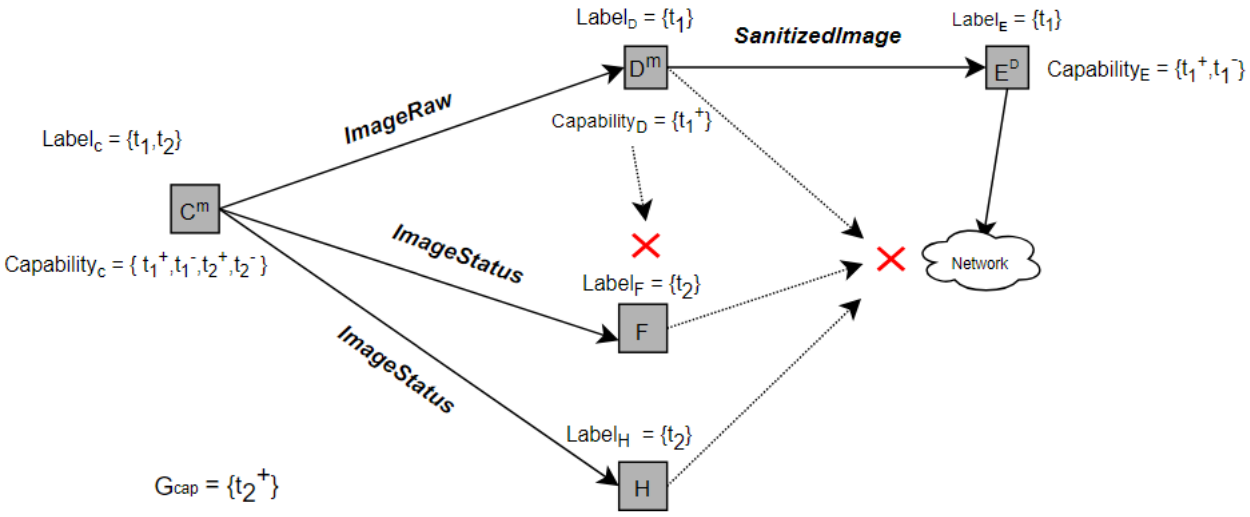


Figure 5.1: Camera DIFC policy

Some of the key challenges that FlowROS solves are listed below:

1. **Data segregation**
   FlowROS's DIFC system enables ROS applications to better control their sensitive data by associating it with the security class and controlling information flow as needed by declaring and defining policies. Also, note that data segregation and controlling require minimal application modifications.

2. **Implicit propagation**
   FlowROS's reference monitor supports implicit label propagation for unmodified applications.

Figure 5.1 shows a practical communication graph of our modified version ROS2 camera application to support DIFC use-case C^m that has two types of data; first, it publishes a piece of sensitive information, a video feed on the topic ImageRaw, and second, comparatively less sensitive information camera status published on the topic ImageStatus. Let us say D^m is a modified image-processing application that requires a video feed and it blurs the sensitive part of the ImageRaw; hence it subscribes to the topic ImageRaw and publishes Sanitized Image further down in the communication graph. E^D is sanitizing application to export sensitive information to the network by declassifying it and hence it is a trusted declassifier. To achieve this, C^m gives t_1^+ capabilities to D^m and E^D (line 5 and 6 in listing 4.1) so that they can

subscribe to the image feed published by C^m and gives E^D t_1^- capability allowing it to declassify data that belong to security class t_1 to communicate to the network. Since D^m and E^D are modified applications and trusted, they follow explicit label propagation to satisfy information flow rules(i.e., applications or nodes are modified at the code level to change their labels). On the other hand, for the less sensitive information (i.e., ImageStatus), C^m adds t_2^+ capability to G_c, implying any application in the system can subscribe to the topic ImageStatus but not having t_2^- will not allow any application to leak the data. The application increments its label before sharing the sensitive information(line 14) and removes tag from its label after sharing sensitive information (line 18).

```
std:: string topic("ImageRaw");      //Sensitive information
std:: string topic("ImageStatus")
create_tag("t1");
create_tag("t2");
add_pos_cap("t1",Image_processing_node); //Image processing app
add_pos_cap_("t1",Declassifier_node);  //trusted declassifier
add_neg_cap("t1",Declassifier_node);
auto node = rclcpp:: node::make_shared("camera");
      // Publish the image message and increment the frame_id.
      . . .
 add_tag_label("t1");
      pub->publish(std::move(msg));
      ++i;
      rclcpp::spin_some(node);
 remove_tag_label("t1");
      . . .
 add_tag_label("t2");
           auto msg_status = std::make_unique<std_msgs::msg::String>();
           msg_status->data = std::to_string(val_status);
            val_status += 1;
            pub_status->publish(std::move(msg_status));
             rclcpp::spin_some(node);
  remove_tag_label("t2");
            delay_status = true;
      . . .
```

Listing 5.1: ROS2 Camera.cpp

5.1 Experiments

We evaluate FlowROS by considering two important questions: 1. What is the impact on the latency involved for communications among applications due to the information flow control checks performed by the reference monitor in the kernel? 2. What system overheads does the application suffer due to DIFC modifications in the application to secure its sensitive information?

We implemented FlowROS on Linux kernel version vanilla 4.9 on Ubuntu 18.04. The choice of this kernel is due to its support for the Nvidia Jetson TX2 board. We used ROS version 2 (dashing) and eProsima FASTRTPS version 1.8.2 [58, 12, 6] that ROS2 integrates for underlying communication protocols among ROS applications. Overall, we provide user space IOCTL interface support for ROS applications to leverage DIFC calls, consisting of 253 lines of code. We implemented FlowROS as an LSM module with an added code of 1575 lines that includes FlowROS's kernel IOCTL interface and the reference monitor responsible for intercepting the communications in the system. We used the Nvidia Jetson TX2 development kit [45] for evaluating the overheads of FlowROS. The Jetson board has quad-core ARM A57 and dual Denver 2 64-bit CPU architecture support, 8GB LPDDR4 RAM, and 32 GB eMMC flash storage support. We evaluated Jetson since the robotics community heavily uses Jetson and similar arm-based boards. The Jetson board also has 256 CUDA computing cores, which are useful for navigation and image-processing software stacks for robotic applications. Although ROS is compatible with any architecture, we use a Nvidia Jetson board with ARM architecture, as it is heavily used for robotic applications [16].

5.1.1 Communication Latency

We use an open-source i-Robot performance evaluation framework [8] designed to evaluate ROS2 applications on different metrics. i-Robot is developed in core C++ language with only library dependency of ROS2. It provides libraries and tools to create applications and evaluate their overheads concerning ROS2 communication, CPU, memory usage, etc. In i-Robot, nodes that run in the context of a process do not perform any computations; therefore, they solely measure overheads. As the reference monitor performs information flow control checks at the process level, this benchmark helps us evaluate the latency incurred at the message level. This benchmark provides a full ROS application environment consisting of 10 and 20 nodes, publishing and subscribing to each other's topics and, in the end, generating a detailed report of overheads incurred. As the nodes publish and subscribe to the topics and do not perform

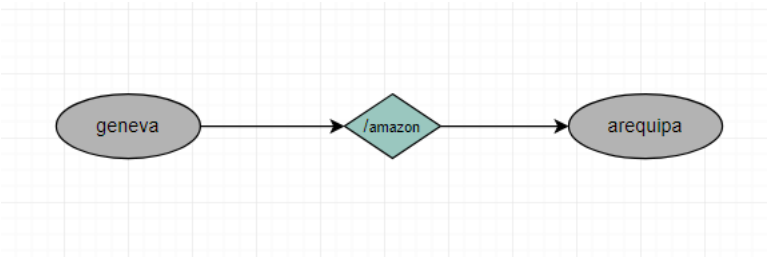


Figure 5.2: Benchmark publish/subscribe example

any active computation, we segregate the entire benchmark consisting of many nodes into two nodes. Still, we configure them on the different parameters (frequency of publishing) to identify the impact on latency due to the reference monitor checks. we modify the i-Robot benchmark to support DIFC calls, such that the user can use the labeling option to run the test with different configurations

Figure 5.2 shows a communication graph between a computing node geneva executing in a process context and publishing on the topic `amazon`, similarly a node arequipa executing in a separate process context subscribes to the topic `amazon`. To observe the communication delay or latency incurred due to DIFC checks, we labeled the processes geneva and arequipa with different label sizes (i.e., the label consists set of tags) and with different publishing frequencies to understand how applications with different publishing frequencies get affected.

i-Robot benchmark samples the resources being used every 500ms for the duration of the test. We ran the above 5.2 benchmark configuration for the duration of 10s for each mean latency entry in the table. The base shows the mean latency observed without labeling the nodes, and Label_1 indicates the mean latency observed with a label size of ten. Similarly, Label_2 and Label_3 indicate mean latencies with label sizes of 100 and 1000, respectively. We observed that labeling applications do not impact overheads in the latency till the frequency of 100. Also, for lower publishing frequencies, DDS buffers the messages the application publishes for a few milliseconds to optimize the data exchange process, therefore incurring a slightly higher mean latency. To justify our hypothesis, we further ran the benchmark 5.2 for a very high-frequency rate so that DDS buffering would be minimized and we could see precise latency.

Table 5.1: Communication latency

Frequency(pub/sec)	Mean (us)			
	Base	Label_1(size=10)	Label_2(size=100)	Label_3(size=1000)
10	4698	5789	5084	5168
50	4460	5003	4277	4780
100	2900	2789	2757	2677

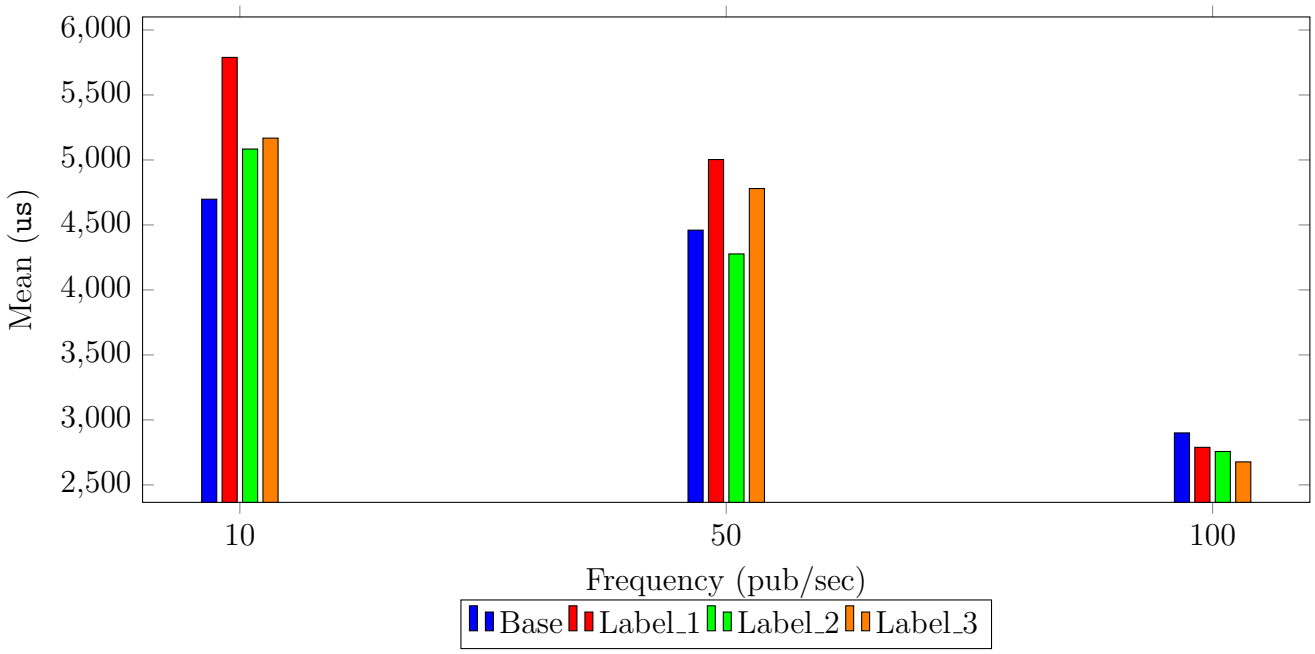


Figure 5.3: Bar Graph of Communication Latency

i-Robot benchmark samples the resources being used every 500ms for the duration of the test. We ran the above 5.2 benchmark configuration for the duration of 10s for each mean latency entry in the table. The base shows the mean latency observed without labeling the nodes, and Label_1 indicates the mean latency observed with a label size of ten. Similarly, Label_2 and Label_3 indicate mean latencies with label sizes of 100 and 1000, respectively.We observed that labeling applications do not show impact overheads in the latency till the frequency of 100. As figure 5.3 represents table 5.1 where for the frequency 10, label size 100 shows lower latency than label size 10 and 1000. Similarly, for frequency 50, communication latency for label size 100 is lower than the base (i.e., without label) and label size 1000. Implying that the deviations in the communication latencies are high. It is also because DDS buffers the messages that the application publishes to optimize the data exchange; hence, for higher publishing frequencies, the average communication latency comes down to around 2600us as the buffering impact will

Table 5.2: Table of mean and minimum latency

Frequency	**Without Label**	
	Mean (us)	**Min (us)**
500	900	459
1000	468	361
Frequency	**With Label**	
	Mean (us)	**Min (us)**
500	921	426
1000	627	297

be lesser. To justify our hypothesis, we ran the benchmark 5.2 for a very high-frequency rate to minimize DDS buffering, resulting in latency till 500us. Table 5.2 shows mean latency numbers for the label size 1000 and publishing frequency 500 and 1000; this results in a slight increase in the mean latencies, but still, the communication overhead due to reference monitor checks is significantly less in comparison with the deviations in the latencies as can be seen in table 5.2 that minimum communication latencies with labeling are still lesser than the base (i.e., without label). Note that it is not the case that with label checks, latencies are always less than without labels; instead, the point is that the deviations in the latencies are higher than the overhead incurred due to reference monitor, and therefore, there is no consistent overhead seen in the numbers even at the high frequencies. To prove that we added a delay of 500 microseconds in the reference monitor code of the kernel as shown in the figure 5.4, precisely where the flow is allowed due to correct label rules, and we saw the delay getting reflected in the numbers. To conclude, label checks performed by the reference monitor incur very low overhead (i.e., in the magnitude of a few microseconds), therefore not impacting much on the overall communication. Note that for evaluation purposes, we tested the latencies for label sizes up to 1000; in practice, label size depends upon the number of security levels the application needs in order to deduce information flow policies (e.g., Label size 10 can represent $2^{10} = 1024$ security levels or classes for its objects and subjects which is more than sufficient [68, 74]).

Table 5.3: Table of mean and minimum latency for added delay of 500us

Frequency	**Without Label**	
	Mean (us)	**Min (us)**
1000	540	330
Frequency	**With Label**	
	Mean (us)	**Min (us)**
1000	1032	715

```
if(dominates(receiver_prov->seclabel,sender_prov->seclabel))
{
    //Flow is allowed because label of receiving process
    //dominates the label of sender.
    udelay(500);
    printk(KERN_INFO"Flow allowed\n");
}
```

Figure 5.4: Added delay in the reference monitor

5.1.2 System Overhead

In the DIFC system, modified applications are required to add a tag or remove a tag from their label depending upon the information flow that the application is going to perform, say increasing the label for a piece of higher classified information and decreasing the label while sharing the information that belongs to a lower class. DIFC delegates these modifications to the application developers to control their sensitive information better. In the previous section, we showed that label checks performed in the kernel do not affect the latency much as they are straightforward and simple instructions to execute. However, on the contrary, label changes require system-level calls (i.e., `ioctls` in our case) that the kernel performs on behalf of an application, therefore incurring some more context switches and more user-kernel overheads. To identify the system-level and performance overhead, the application might suffer, we use the same benchmarking application 5.2 as the i-Robot framework also samples the system-level performance overheads incurred by the applications while running. In addition to that, we use `strace` and `time` tools available on the Linux machine to identify user-space and kernel-space overheads at the run-time due to application modification. We configure `strace` with an option to timestamp the time taken by each `ioctl` call that the application has performed for label changes. Moreover, it generates the report of the total kernel-space execution time. We modified the i-Robot benchmark to change its label using FlowROS's `ioctl` interface.

Table 5.4: System-level overheads

Frequency	**Without Label**			
	No. of IOCTL	**Userspace (s)**	**Kernelspace (s)**	**CPU Utilization (%)**
1	-	0.342	0.260	0.592
5	-	0.818	0.465	0.890
10	-	1.014	0.549	0.910
50	-	2.132	1.539	2.5
100	-	3.984	2.974	4.01
Frequency	**With Label**			
	No. of IOCTL	**Userspace (s)**	**Kernelspace (s)**	**CPU Utilization (%)**
1	600	0.360	0.306	0.689
5	3000	0.893	0.617	1.21
10	6000	1.358	0.832	1.649
50	30 000	2.941	3.346	4.34
100	60 000	4.066	4.087	4.96

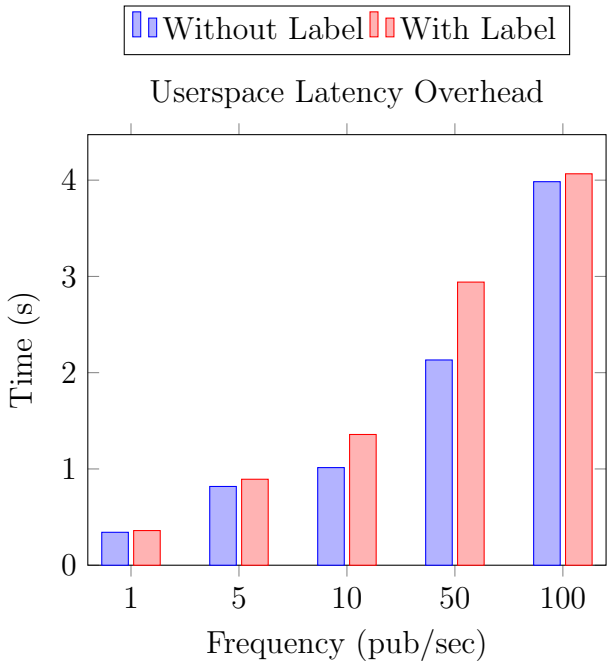


Figure 5.5: Userspace Latency Comparison

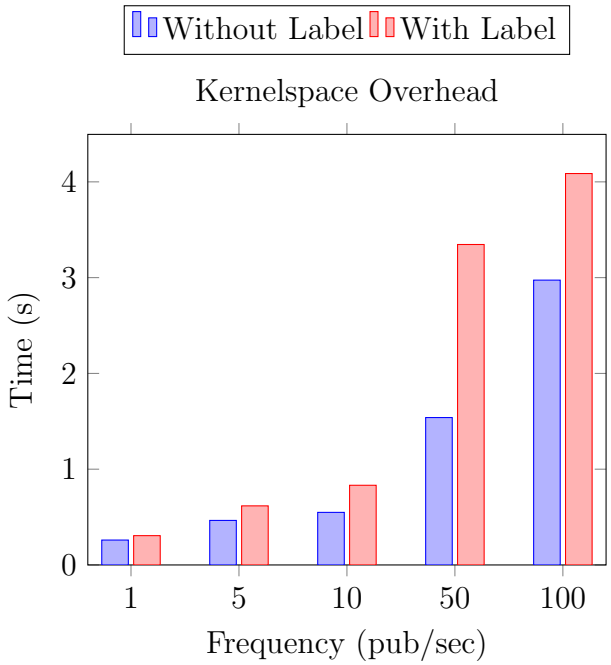


Figure 5.6: Kernelspace Latency Comparison

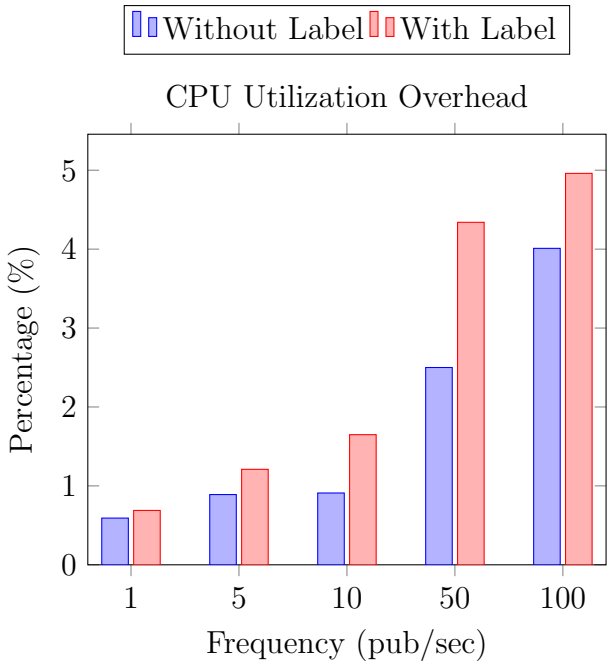


Figure 5.7: Comparison of CPU Utilization: Without Label vs. With Label

In Figure 5.2, the Geneva application publishes messages on the topic "Amazon," and we analyzed the system overheads on the Geneva node. For this analysis, we added a set of 10 tags to its label before publishing anything and removed the tags from its label after publishing. This methodology was chosen to observe the system-level overheads incurred due to application modifications at runtime. This experiment mimics the practical behavior of applications that use DIFC calls to protect their sensitive information. Table 5.4 and Figures 5.5, 5.6, and 5.7 show the overheads incurred in the application due to DIFC modifications. The frequency represents the rate at which the application publishes messages per second, adding the label before publishing the message and removing it afterward. It is important to note that the system overhead shown in Figures 5.5 and 5.6 represents the amount of time spent by the application in the Userspace and Kernelspace. With labeling, the application does show an increase in Userspace time, but the kernel overhead is slightly higher than that of the Userspace. This is because changing labels requires the application to make IOCTL calls, leading to context switches, which are considerably heavier. Hence, the kernel overhead is more evident in Figure 5.6. Applications must open the FlowROS `ioctl` device `/dev/flowros` and send label change `ioctl` calls at runtime to alter their labels.

5.1.3 Availability

Our implementation source and benchmarks are available at this URL[1]: https://github.com/ChinmayGameti/FlowROS

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis has presented FlowROS, a Decentralized Information Flow Control (DIFC) system tailored for the Robot Operating System (ROS), addressing the critical security and privacy challenges inherent in the existing ROS framework. FlowROS presents an OS-level DIFC mechanism, empowering developers to safeguard sensitive information more effectively. We have identified the limitations of Mandatory Access Control MAC systems in the context of ROS, as these system's coarse granularity and complex policy deduction requirements often result in inefficient security measures for robotic platforms. The dual approach of explicit and implicit label propagation in FlowROS ensures backward compatibility with unmodified ROS applications while maintaining robust information flow control. FlowROS effectively tackles the inherent DIFC challenge of label explosion in ROS, ensuring the system remains functional and secure. Implementing FlowROS is demonstrated through practical policies and a case study on an existing ROS application.

6.2 Future Work

- The Robot Operating System (ROS) facilitates domain separation by enabling virtual isolation for applications running on the same system but operating in distinct domain planes. This architectural design ensures that applications within the same domain can exclusively discover and communicate with each other, effectively segregating their operational environments. However, this domain separation paradigm, integral to ROS, is not inherently mirrored in Information Flow Control (IFC) systems at the operating system (OS) level. In the OS context, there is a lack of visibility regarding which processes and objects belong to specific domains, leading to potential challenges in policy enforcement

and information security.

A promising direction for future development involves extending the domain separation concept to OS-level IFC systems. This IFC systems could robustly enforce domain-specific policies by integrating domain ID information into the OS-level reference monitor. This enhancement would allow the OS to recognize and respect the domain boundaries established in ROS, thus providing a more cohesive and secure environment for applications. Such an integration would not only reinforce the isolation between different domains but also streamline the management of information flow and access control, adhering to each domain's specific needs and constraints.

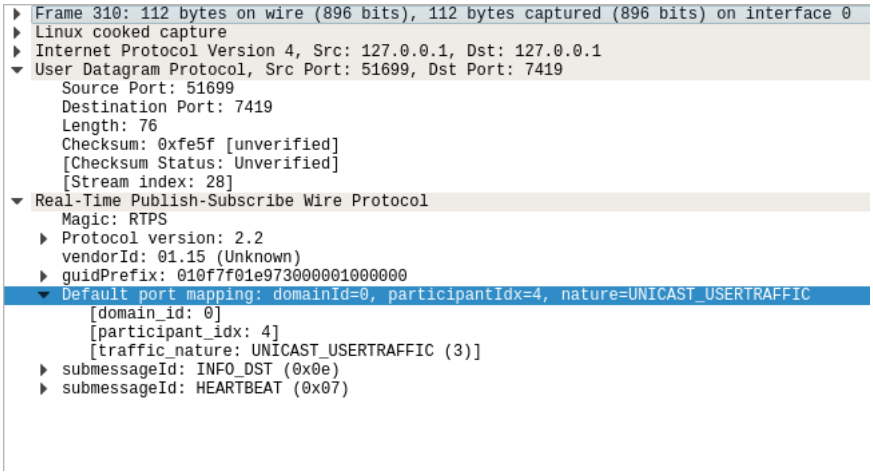
- FlowROS, as an OS-level DIFC system, effectively secures sensitive data within single-system ROS applications. However, its current design is limited to individual systems and doesn't extend to distributed ROS networks. A key area for future enhancement is the adaptation of FlowROS for distributed environments. This advancement would significantly improve data protection across interconnected ROS systems, meeting the evolving demands of complex, networked robotic applications.

- Similar to RIFLE [67] and Loki [75], which showcase Information Flow Control (IFC) systems combining secure kernel and hardware support, a parallel approach could be beneficial for the ROS environment. Exploring this direction could enhance the security framework within ROS.

Appendix A

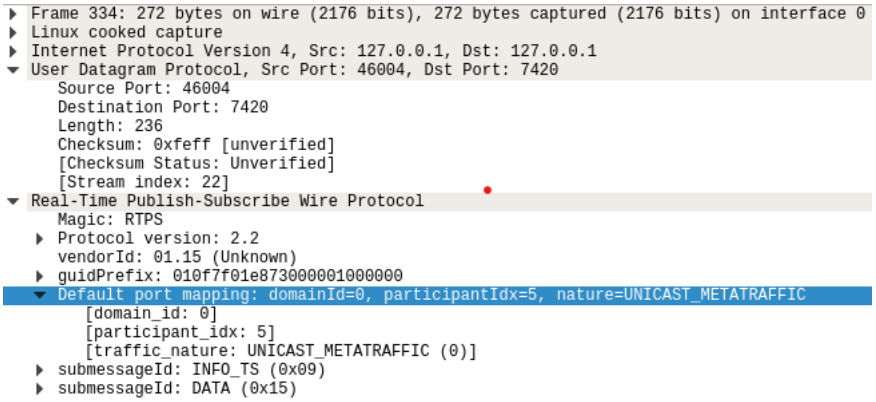
RTPS Traffic

A.1 User traffic



This type of traffic refers to the actual data being exchanged between the nodes in a ROS platform. It consists of the messages that are published and subscribed to by different nodes.

A.2 Metadata Traffic



- This type of traffic, on the other hand, pertains to the communication related to the metadata about the DDS entities.
- Metadata traffic includes information such as announcements of new data writers or readers, status messages, and other control or discovery information necessary for the establishment and maintenance of the DDS communication.
- This traffic is essential for the underlying DDS middleware to manage the network of distributed nodes, ensuring that publishers and subscribers are correctly connected and synchronized.

Bibliography

[1] Flowros. https://github.com/ChinmayGameti/FlowROS. 53

[2] ROS1 Technical overview. http://wiki.ros.org/ROS/Technical%20Overview. 7

[3] App-armor is an effective and easy-to-use linux application security systems. https://example.com/app-armor. 2, 25, 26

[4] Cyclone dds. https://example.com/cyclone-dds. 1, 9

[5] Data distribution service (dds). https://www.omg.org/spec/DDS/1.4/PDF, . 1, 9

[6] The real-time publish-subscribe protocol dds interoperability wire protocol. https://www.omg.org/spec/DDSI-RTPS/2.3/PDF, . 1, 9, 22, 46

[7] eprosima. https://example.com/eprosima. vii, 9, 10

[8] irobot ros 2 performance evaluation framework. https://github.com/irobot-ros/ros2-performance. 46

[9] Omg ddsi-rtps specification. https://www.omg.org/spec/DDSI-RTPS/2.2, . 9, 11

[10] Omg ddsi-rtps specification. https://www.omg.org/spec/DDSI-RTPS/2.5/PDF, . 1, 9, 11, 13, 22

[11] Open dds. https://example.com/open-dds. 1

[12] ROS2 Client libraries, . https://index.ros.org/doc/ros2/Concepts/ROS-2-Client-Libraries/#common-functionality-the-rcl. 1, 7, 46

[13] Vortex open slice. https://example.com/vortex-open-slice. 9

[14] Jean Bacon, David M Eyers, Jatinder Singh, and Peter R Pietzuch. Access control in publish/subscribe systems. In *Proceedings of the second international conference on Distributed event-based systems*, pages 23–34, 2008. 25

[15] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. Run-time monitoring and formal analysis of information flows in chromium. In *NDSS*, 2015. 38

[16] Rakesh Rajan Beck, Abhishek Vijeev, and Vinod Ganapathy. Privaros: A framework for privacy-compliant delivery drones. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 181–194, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370899. doi: 10.1145/3372297.3417858. URL https://doi.org/10.1145/3372297.3417858. 2, 16, 18, 19, 20, 25, 38, 42, 46

[17] William E Boebert and R Y Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, 1985. 13

[18] David F C Brewer and Michael J Nash. The chinese wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1989. doi: 10.1109/SECPRI.1989.36295. 13

[19] Kun Cheng, Yuan Zhou, Bihuan Chen, Rui Wang, Yuebin Bai, and Yang Liu. Guardauto: A decentralized runtime protection system for autonomous driving. *IEEE Transactions on Computers*, 70(10):1569–1581, 2021. doi: 10.1109/TC.2020.3018329. 1, 18, 19

[20] David D Clark and David R Wilson. A comparison of commercial and military computer security policies. In *1987 IEEE Symposium on Security and Privacy*, pages 184–184. IEEE, 1987. 31

[21] Gelei Deng, Guowen Xu, Yuan Zhou, Tianwei Zhang, and Yang Liu. On the (in) security of secure ros2. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 739–753, 2022. 2

[22] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5): 236–243, may 1976. ISSN 0001-0782. doi: 10.1145/360051.360056. URL https://doi.org/10.1145/360051.360056. 16, 26

[23] Bernhard Dieber, Stefan Kacianka, Stefan Rass, and Peter Schartner. Application-level security for ros-based applications. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 4477–4482. IEEE, 2016. 1, 7, 9, 18, 19, 26

[24] Bernhard Dieber, Bernhard Breiling, Sebastian Taurer, Stefan Kacianka, Stefan Rass, and Peter Schartner. Security for the robot operating system. *Robotics and Autonomous Systems*, 98, 2017. 9, 19, 26

[25] David F Ferraiolo, RD Kuhn, and Ramaswamy Chandramouli. Role-based access control, 2nd edn. artech house. *Inc., Norwood*, 2007. 25

[26] Blake Hannaford, Jacob Rosen, Diana W. Friedman, Hawkeye King, Phillip Roan, Lei Cheng, Daniel Glozman, Ji Ma, Sina Nia Kosari, and Lee White. Raven-ii: An open platform for surgical robotics research. *IEEE Transactions on Biomedical Engineering*, 60 (4):954–959, 2013. doi: 10.1109/TBME.2012.2228858. 1

[27] Michael A Harrison, Walter L Ruzzo, and Jeffrey D Ullman. Protection in operating systems. *Communications of the ACM*, August 1976. doi: 10.1145/360303.360333. 14

[28] Tanmay Jain and Gene Cooperman. Dmtcp: Fixing the single point of failure of the ros master. In *ROSCON 2017: the ROS Developers Conference*, 2017. 1, 9

[29] Se-Yeon Jeong, I-Ju Choi, Yeong-Jin Kim, Yong-Min Shin, Jeong-Hun Han, Goo-Hong Jung, and Kyoung-Gon Kim. A study on ros vulnerabilities and countermeasure. In *Proceedings of the Companion of the 2017 ACM/IEEE International Conference on Human-Robot Interaction*, pages 147–148, 2017. 24

[30] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. Run-time enforcement of information-flow properties on android. In *Computer Security–ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings 18*, pages 775–792. Springer, 2013. 27, 33

[31] Hwimin Kim, Dae-Kyoo Kim, and Alaa Alaerjan. Abac-based security model for dds. *IEEE Transactions on Dependable and Secure Computing*, 19(5):3113–3124, 2021. 25

[32] Maxwell Krohn and Eran Tromer. Noninterference for a practical difc-based operating system. In *2009 30th IEEE Symposium on Security and Privacy*, pages 61–76. IEEE, 2009. 26

[33] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. *SIGOPS Oper. Syst. Rev.*, 41(6):321–334, oct 2007. ISSN 0163-5980. doi: 10.1145/1323293.1294293. URL https://doi.org/10.1145/1323293.1294293. 3, 16, 17, 26, 28, 30, 32, 34, 40

[34] Sven Lachmund. Auto-generating access control policies for applications by static analysis with user input recognition. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, page 8–14, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589657. doi: 10.1145/1809100.1809102. URL https://doi.org/10.1145/1809100.1809102. 3, 16

[35] Yun Li, Chenlin Huang, Lu Yuan, Yan Ding, and Hua Cheng. Aspgen: an automatic security policy generating framework for apparmor. In *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 392–400, 2020. doi: 10.1109/ISPA-BDCloud-SocialCom-SustainCom51426.2020.00075. 3, 16

[36] Victor Mayoral, Alejandro Hernández, Risto Kojcev, Iñigo Muguruza, Irati Zamalloa, Asier Bilbao, and Lander Usategi. The shift in the robotics paradigm — the hardware robot operating system (h-ros); an infrastructure to create interoperable robot components. In *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 229–236, 2017. doi: 10.1109/AHS.2017.8046383. 24

[37] Giovanni Mazzeo and Mariacarla Staffa. Tros: Protecting humanoids ros from privileged attackers. *International Journal of Social Robotics*, 12:827–841, 2020. 25

[38] Jarrod McClean, Christopher Stull, Charles Farrar, and David Mascareñas. A preliminary cyber-physical security assessment of the Robot Operating System (ROS). In Robert E. Karlsen, Douglas W. Gage, Charles M. Shoemaker, and Grant R. Gerhart, editors, *Unmanned Systems Technology XV*, volume 8741, page 874110. International Society for Optics and Photonics, SPIE, 2013. doi: 10.1117/12.2016189. URL https://doi.org/10.1117/12.2016189. 2

[39] John McClean, Christopher Stull, Charles Farrar, and David Mascareñas. A preliminary cyber-physical security assessment of the robot operating system (ros). In *Unmanned Systems Technology XV*, volume 8741, page 874110. International Society for Optics and Photonics, 2013. 9, 24

[40] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, page 228–241, New York, NY, USA, 1999. Association for

Computing Machinery. ISBN 1581130953. doi: 10.1145/292540.292561. URL https://doi.org/10.1145/292540.292561. 17

[41] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31(5):129–142, oct 1997. ISSN 0163-5980. doi: 10.1145/269005.266669. URL https://doi.org/10.1145/269005.266669. 17

[42] Andrew C Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. software release, 2001. 17, 28

[43] Adwait Nadkarni and William Enck. Preventing accidental data disclosure in modern operating systems. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 1029–1042, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324779. doi: 10.1145/2508859.2516677. URL https://doi.org/10.1145/2508859.2516677. 16, 27, 33, 34

[44] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. Practical {DIFC} enforcement on android. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1119–1136, 2016. 3, 4, 18, 27, 28, 29, 30, 32, 33, 34, 38, 40

[45] I NVIDIA and TX Jetson. developer kit and modules, 2021. *URL https://www. nvidia. com/en-us/autonomous-machines/embedded-systems/jetson-tx2*. 46

[46] Object Management Group (OMG) Std. ptc/17-09-20. DDS Security. http://www.omg.org/spec/DDS-SECURITY/1.1/, 2017. Rev. 1.1, Sept. 2017. vii, 2, 19, 20

[47] OMG. eprosima. https://fast-dds.docs.eprosima.com/en/latest/fastdds/library_overview/library_overview.html. vii, 10

[48] Ioannis Papagiannis, Pijika Watcharapichat, Divya Muthukumaran, and Peter Pietzuch. Browserflow: Imprecise data flow tracking to prevent accidental data disclosure. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343008. doi: 10.1145/2988336.2988345. URL https://doi.org/10.1145/2988336.2988345. 16, 26

[49] Gerardo Pardo-Castellote and Gabriela Cretu-Ciocarlie. Securing access to distributed pub-sub information in a system-of-systems and gig environment. *Real-Time Innovations, Inc.(May 2010)*, 2010. 2

[50] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. SoCC '17, page 405–418, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350280. doi: 10.1145/3127479.3129249. URL https://doi.org/10.1145/3127479.3129249. 3, 16

[51] Lauri IW Pesonen, David M Eyers, and Jean Bacon. Access control in decentralised publish/subscribe systems. *J. Networks*, 2(2):57–67, 2007. 25

[52] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009. 8

[53] Tobias Rauter, Andrea Höller, Nermin Kajtazovic, and Christian Kreiner. Towards an automated generation of application confinement policies with binary analysis. In *2015 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–6, 2015. doi: 10.1109/ISNCC.2015.7238568. 3, 16

[54] Sean Rivera and Radu State. Securing robots: An integrated approach for security challenges and monitoring for the robotic operating system (ros). In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 754–759, 2021. 2, 24, 26

[55] Sean Rivera, Sofiane Lagraa, and Radu State. Rosploit: Cybersecurity tool for ros. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 415–416. IEEE, 2019. 26

[56] Joseph M. Romano, Jordan P. Brindza, and Katherine J. Kuchenbecker. Ros open-source audio recognizer: Roar environmental sound detection tools for robot programming. *Auton. Robots*, 34(3):207–215, apr 2013. ISSN 0929-5593. doi: 10.1007/s10514-013-9323-6. URL https://doi.org/10.1007/s10514-013-9323-6. 1

[57] ros. ROS.org—Powering the World's Robots, . https://www.ros.org. 1, 7

[58] ros2. ROS 2–ROS 2 documentation, the latest version of the robot operating system. https://index.ros.org/doc/ros2/. 1, 7, 42, 46

[59] Matthew Rossi, Dario Facchinetti, Enrico Bacis, Marco Rosa, and Stefano Paraboschi. SEApp: Bringing mandatory access control to android apps. In *30th USENIX Se-*

curity Symposium (USENIX Security 21), pages 3613–3630. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/rossi. 3, 16, 25, 26

[60] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. *SIGPLAN Not.*, 44(6):63–74, jun 2009. ISSN 0362-1340. doi: 10.1145/1543135.1542484. URL https://doi.org/10.1145/1543135.1542484. 3, 17, 18, 25, 26, 28, 30, 32, 34

[61] RTI. *RTI Connext DDS Core Libraries and Utilities, User Manual*. RTI, 2013. Version 5.1.0. 1, 9

[62] John Rushby. The bell and la padula security model. *Computer Science Laboratory, SRI International, Menlo Park, CA*, 1986. 29, 31, 32

[63] R.S. Sandhu and P. Samarati. Access control: principle and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994. doi: 10.1109/35.312842. 26

[64] Gerhard Schellhorn, Wolfgang Reif, Axel Schairer, Paul A Karger, Volker Austel, and Daniel Toll. Verification of a formal security model for multiplicative smart cards. In *Proceedings of the European Symposium on Research in Computer Security*, 2000. 13

[65] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001. 25, 26

[66] Deian Stefan, Edward Z Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazieres. Protecting users by confining {JavaScript} with {COWL}. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 131–146, 2014. 38

[67] N. Vachharajani, M.J. Bridges, J. Chang, R. Rangan, G. Ottoni, J.A. Blome, G.A. Reis, M. Vachharajani, and D.I. August. Rifle: An architectural framework for user-centric information-flow security. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 243–254, 2004. doi: 10.1109/MICRO.2004.31. 17, 18, 55

[68] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4):11–es, dec 2007. ISSN 0734-2071. doi: 10.1145/1314299.1314302. URL https://doi.org/10.1145/1314299.1314302. 3, 17, 26, 28, 30, 32, 33, 49

[69] Ruowen Wang, William Enck, Douglas S. Reeves, Xinwen Zhang, Peng Ning, Dingbang Xu, Wu Zhou, and Ahmed M. Azab. Easeandroid: Automatic policy analysis and refinement for security enhanced android via large-scale semi-supervised learning. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 351–366. USENIX Association, 2015. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wang-ruowen. 3, 16

[70] Ruowen Wang, Ahmed M. Azab, William Enck, Ninghui Li, Peng Ning, Xun Chen, Wenbo Shen, and Yueqiang Cheng. Spoke: Scalable knowledge collection and attack surface analysis of access control policy for security enhanced android. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 612–624, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349444. doi: 10.1145/3052973.3052991. URL https://doi.org/10.1145/3052973.3052991. 16

[71] Robert White, Henrik I Christensen, and Morgan Quigley. Sros: Securing ros over the wire, in the graph, and through the kernel. *ArXiv e-prints*, 1611.07060, 2016. 2, 18, 24

[72] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *11th USENIX Security Symposium (USENIX Security 02)*, San Francisco, CA, August 2002. USENIX Association. URL https://www.usenix.org/conference/11th-usenix-security-symposium/linux-security-modules-general-security-support-linux. 4, 17, 33

[73] Yuanzhong Xu and Emmett Witchel. Maxoid: Transparently confining mobile applications with custom views of state. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–16, 2015. 27

[74] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, page 19, USA, 2006. USENIX Association. 3, 17, 26, 28, 32, 49

[75] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the*

8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, page 225–240, USA, 2008. USENIX Association. 17, 18, 55