Decentralized Information-Flow Control for ROS2

Nishit V. Pandya, Himanshu Kumar, Gokulnath M. Pillai, Vinod Ganapathy
Department of Computer Science and Automation
Indian Institute of Science, Bangalore, India
{nishitv,himanshukuma,gokulnathp,vg}@iisc.ac.in

***Abstract*—ROS2 is a popular publish/subscribe based middleware that allows developers to build and deploy a wide-variety of distributed robotics applications. Unfortunately, ROS2 offers applications poor control over how their data is consumed downstream by other applications. Although decentralized information-flow control (DIFC) offers a solution to this problem, the decentralized and distributed architecture of ROS2 poses new challenges to building a practical DIFC system for ROS2.**

We present Picaros, a DIFC system tailored for ROS2. Picaros adopts a novel approach to DIFC that casts and solves DIFC's access control problem in the framework of attribute-based encryption (ABE). Picaros's design embraces the unique nature of the ROS2 platform and carefully avoids any centralized elements. This paper presents the design and implementation of Picaros and reports results from our experiments that use Picaros's ABE-based approach for DIFC with ROS2 applications.

I. Introduction

The Robot Operating System (version 2), or ROS2 [58], is a popular middleware framework that is used by several robotics platforms. ROS2 provides a convenient API that allows applications to execute and communicate over a collection of robots. The popularity of ROS2 has led to an active developer community and a thriving market for ROS2-based apps, with many robotics vendors adopting a ROS2-based stack for their offerings (*e.g.,* Amazon Robomaker [29], iRobot [2]).

ROS2 uses publish/subscribe as the primary method of communication. ROS2 applications publish or subscribe to a set of *topics*, and messages in the system are tagged by topic. Each application advertises the set of topics to which it publishes or subscribes. ROS2 uses a decentralized approach (building atop the Data Distribution Service, or DDS [23, 35, 60]) to identify publishers and subscribers that must be matched up by topic name, and establishes a communication channel between them. For example, a `Camera` application may publish to a topic Image, to which the applications `Navigator` and `ObjectDetector` subscribe. ROS2 sets up pairwise connections between `Camera`/`Navigator` and `Camera`/`ObjectDetector` and the applications communicate directly.

Secure ROS2 (SROS2) [46, 62] extends ROS2 with basic security features. With SROS2, each application provides a manifest in which it declares the topics to which it publishes or subscribes. SROS2 cryptographically binds applications and their manifests, and during application startup, sandboxes the application to only publish or subscribe to the topics advertised in the manifest. SROS2 encrypts all application communication end-to-end using TLS, thereby providing confidentiality, integrity, and sender authentication for inter-application message exchange. Because these security features are so foundational, for the rest of this paper, we will assume the presence of SROS2 as a default in the ROS2 stack.

Unfortunately, the security features of SROS2, while essential, do not go far enough to provide applications fine-grained control over how their data is used by downstream applications. For example, when `ObjectDetector` consumes a message tagged with the topic Image, it is free to publish this image (or information about the image) under a different topic name. ROS2 does not provide the data owner that publishes to the topic Image (*e.g.,* the `Camera` application) with any primitives that allow it to exert downstream control over how this data is used. This lack of downstream control can lead to data exfiltration attacks that may not be acceptable to the application that owned/created the data.

This problem has been extensively studied in the security community, which has developed the theory [49–52] and practical systems (*e.g.,* [30, 40, 43, 44, 53, 74]) based on decentralized information-flow control (DIFC) as a solution. In a DIFC-based system, each data owner associates the data objects they own/create with a label, which is a set of tags of the data owner's choosing. DIFC-enforcement controls how a data object is consumed by checking that the label of a data consumer has all the tags in the data object's label.

This paper presents **Picaros**[*], a DIFC system for ROS2. Picaros leverages the novel insight that ROS2 application authors already tag the messages an application publishes with topics that are semantically associated with the content of the message. Picaros bootstraps DIFC labels using ROS2 topics, thus both reducing additional developer effort needed to specify DIFC labels and also ensuring that the DIFC labels semantically reflect the type of data being protected.

A key design principle that we used in building Picaros is to *respect the distributed and decentralized nature of ROS2.* ROS2 is designed to support applications on distributed robotics platforms. Distributed ROS2 applications execute on a number of different hardware platforms, and implement their end-to-end functionality using seamless communication enabled by ROS2. To support such distributed applications and allow them to scale, ROS2 consciously adopts decentralization, and meticulously avoids any centralized elements in its design. This is a key feature that differentiates ROS2 from its predecessor, ROS1 [55, 57] as well as other popular publish/subscribe

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24101
www.ndss-symposium.org

[*]Picaros is Information-flow Control via ABE for ROS2.

systems such as MQTT [32] that use a centralized broker. For example, ROS2 eliminates a centralized matchmaking service (the `rosmaster` [57]) by replacing it with DDS's decentralized discovery protocol. ROS2 is also dynamic in that it allows new distributed applications to spawn up at any time and seamlessly communicate with other applications running on the system.

Prior DIFC systems have focused on enforcing DIFC policies atop a single host, with DIFC policy enforcement implemented within the operating system (OS) (*e.g.,* [30, 40, 74]), application runtime (*e.g.,* [19, 52]), or middleware [53]. In a distributed setting, DIFC policies must be enforced on applications running across a collection of machines. This requires a policy enforcement infrastructure that spans these machines, and a way to securely bind DIFC labels to the data objects exchanged between the machines. Systems such as Fabric [43, 44] address DIFC in a distributed setting, but require applications to be written in a customized language, with the language runtime responsible for enforcing DIFC policies. DStar [75] enforces DIFC policies on UNIX processes in a distributed setting by running a DIFC-capable OS on each machine (HiStar [74]). DStar uses a dedicated data exporter process on each machine, which holds sole responsibility for network access (*i.e.,* it is the only entity that sends data objects over the network) and for binding DIFC-labels to exported and received data objects. Designing a DIFC system for ROS2 using a centralized data exporter per machine, which would be common to all applications running on that machine, would directly violate ROS2's design philosophy of decentralization.

Picaros instead uses a novel approach that leverages *attribute-based encryption* (ABE) [12] to bind DIFC labels to data objects and enforce DIFC policies in a distributed setting. In ABE, the cryptosystem associates a user's decryption key with a set of attributes, *e.g.,* represented as strings. Ciphertexts in ABE are associated with an access structure over these attributes, *e.g.,* a Boolean expression over attributes. The cryptosystem is set up so that the user can decrypt a ciphertext only if the decryption key's attributes pass through the ciphertext's access structure, *e.g.,* if the corresponding Boolean expression evaluates to `True`. We observe that the features of ABE lend themselves well to enforcing DIFC. Data objects can be encrypted with an access structure that encodes access control rules using the DIFC label associated with that object. Picaros is set up so that an application gets a decryption key based upon the DIFC labels that it possesses. Thus, only applications that have the labels to access a data object on a traditional DIFC system will be able to successfully decrypt and obtain clear-text access to that data object. We explain our ABE encoding of DIFC in Section III.

Picaros's use of ABE to enforce DIFC has two benefits:

► First, it offers a robust way to bind DIFC labels with data objects for distributed settings. The sender of the data object encrypts the object and uses the ABE access structure to encode label comparison with the object's DIFC label, thus ensuring that neither the data object nor the label can be maliciously modified during network transmission. Importantly, senders perform this encoding themselves, thus eliminating the need for a dedicated data exporter, as in DStar [75]. Receivers get keys associated with their DIFC labels, ensuring that only authorized readers can access the data object.

► Second, it provides a convenient way to implement DIFC policy enforcement atop ROS2. DIFC policies can also be enforced via sender-side access control, in which the publisher checks the label of each potential recipient against that of the message to determine whether that recipient can receive the message within the confines of the DIFC label system. Implementing this approach would require invasive changes to the node discovery protocol in ROS2, with a trusted entity that can vouch for the DIFC label of each potential recipient (see Section II for details). Such an entity would have to manage the DIFC label state of every participating application. Having such an entity is against the decentralized ethos of ROS2. Picaros's approach of using ABE for DIFC enforcement overcomes all these problems.

ABE lends itself to a fully decentralized implementation. Early versions of ABE required a centralized master key server [12], but subsequent refinements relaxed the need for such a server under various settings (*e.g.,* [16, 21, 22, 42]). Picaros uses the Lewko-Waters fully collusion-resistant, decentralized multi-authority ABE [42] that elides the need for a centralized master key server. This ABE protocol also works in dynamic settings, allowing new participants to engage immediately with other participants in the system, without elaborate steps for setup. Picaros is thus decentralized and robust in a distributed setting and allows dynamism in application participation, in keeping with ROS2's design philosophy.

To our knowledge, Picaros is the first DIFC system proposed for ROS2. We discuss how Picaros implements its ABE based approach atop the ROS2 software stack in Section IV, and experimental results with our Picaros-prototype applied to various ROS2 benchmarks in Section V and VI. We close with a discussion of related work in Section VII.

II. Background and Overview

In this section, we motivate the problem and our approach using Figure 1 as an example. Along the way, we introduce concepts relevant to ROS2 and DIFC. Figure 1 shows an example of an application pipeline on a ROS2-based platform, such as a drone or a robot. Each oval in the figure represents an application, while arrows are labeled with ROS2 topic names, showing publish/subscribe relationships between applications. This application pipeline is inspired by similar real-world ones, such as NVidia's Isaac ROS pipelines [54].

A camera application (`Camera` in Figure 1) on the platform captures raw images, and publishes them under the topic ImageRaw,[†] to which `FormatConvertor` subscribes. In turn, this application processes the image and publishes both high-resolution and low-resolution formats of the image. The `DNNEncoder` application starts an application pipeline that uses images published under the ImgHiRes topic for navigation, while those published under ImgLoRes are sent to a `Logger` application which stores these in local storage for later retrieval. Each application encodes business logic, and uses ROS2's API to communicate with other applications by suitably publishing or subscribing to specific topics. ROS2 is implemented as a shared library that each application links against. Internally, ROS2 uses the Data-Distribution Service

[†]Throughout the paper, we use different font faces (as illustrated here) to refer to: ROS2 Topics, `Applications` and DIFC labels.

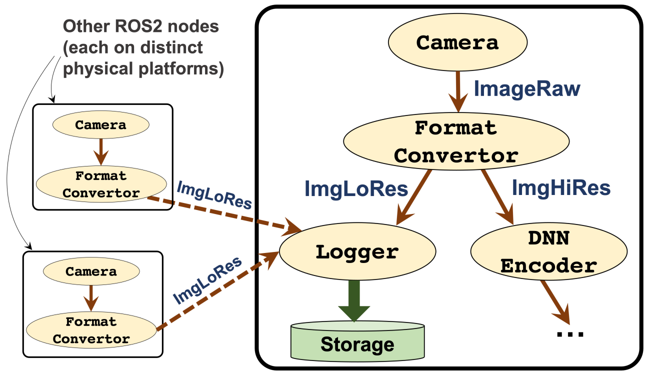


Figure 1: Running example of a ROS2 application pipeline.

(DDS) [35] for matchmaking applications based on topic name and to facilitate communication between them. We assume that all communication between applications is secured end-to-end with TLS—SROS2 enhances ROS2 to provide this feature. Robotics platforms are often distributed in nature, and ROS2 facilitates easy communication across a collection of ROS2 platforms. Figure 1 also illustrates this situation, wherein `Logger` receives messages published by the `FormatConvertor` application running on other ROS2 platforms as well.

Unfortunately, on publish/subscribe systems such as ROS2, applications lose downstream control over their data. For example, consider the `Camera` application, which may wish to impose constraints over how the images that it produces are consumed. While `Camera` is aware of "next hop" applications such as `FormatConvertor` that consume its data, it cannot control how this application publishes the data. `Camera` may wish to enforce that the images published under the topic ImgLoRes are suitably scrubbed to remove privacy-sensitive content before they are consumed by `Logger`. It may not trust `FormatConvertor` to perform such scrubbing, and may wish to ensure that the images are scrubbed by an application that it trusts (*e.g.,* `ImgScrubber` in Figure 2). However, ROS2 (and SROS2) do not provide any mechanisms for `Camera` to exert such downstream control over its data.

Prior work has proposed system-wide mandatory access control (MAC) mechanisms atop ROS2 to improve applications' control over their data [5, 10]. These systems enforce policies by analyzing the flow graph implied by the publish/subscribe relationship between applications, and allow or deny certain applications from communicating with each other at the system level. However, these systems work best only when the set of all applications that will execute on the platform is known *a priori*. Such a static approach is unfortunately a poor fit for dynamic environments—ROS2 applications may be installed and executed on the fly. Moreover, MAC policies are generally expressed as system-wide policies, and set by a system administrator. Individual applications do not have much flexibility in deciding how their data is consumed.

A. DIFC to enforce control over data

DIFC [30, 40, 43, 44, 49–53, 74] overcomes many of the limitations of traditional MAC-based systems by making policy specification *egalitarian*. In a DIFC system, data objects each have an associated *DIFC label* (henceforth, simply *label*), which is a set of *tags*. Each application can include its own tags in the label of data objects that it creates. Intuitively, tags can be used to encode the provenance of a data object, and therefore indicate the security level associated with that data object. DIFC systems generally have separate labels for secrecy and integrity, each with their own set of tags, but for the purposes of this paper, we only focus on secrecy labels. Applications on a DIFC system are also associated with labels. The DIFC system ensures that the tags in the application's label are inherited and included in the label of data objects that the application creates, with the exception of the tags that the application itself creates. For such tags, the application chooses which tags it wants to attach to the data object.

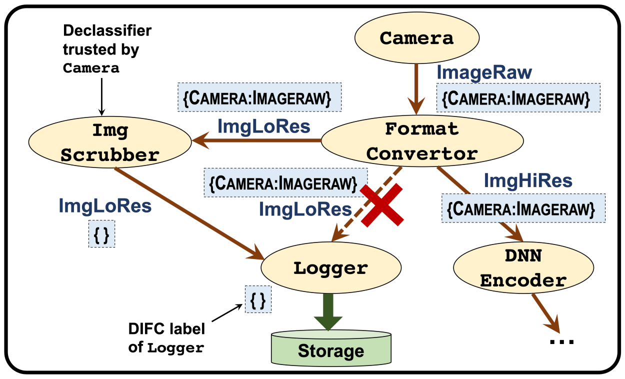
Considering the example in Figure 1, suppose that the `Camera` application initially has a label $\mathsf{L}_{\text{Cam}} = \phi$ (*i.e.,* the empty set without any tags), and that this application decides to attach a tag CAMERA:IMAGERAW to the label of its output (see also Figure 2). The resulting data objects will have the label $\mathsf{L}_{\text{Img}} = \{\text{CAMERA:IMAGERAW}\}$. The main goal of DIFC systems is to use labels to control how data flows within the system—the label of a data consumer must include all of the tags in the label of the data object that it consumes. Thus, for the application `FormatConvertor` to consume the output of `Camera`, its label L_{FC} must satisfy $\mathsf{L}_{\text{Img}} \subseteq \mathsf{L}_{\text{FC}}$. We will henceforth refer to this access control rule as the *no-reads-up* rule [11]. Dually, if an application publishes data, then the data's DIFC label will inherit the tags of the publisher—the *no-writes-down* rule [11].

DIFC systems differ in how they deal with label mismatches that disallow flows according to the rule above. In a DIFC system in which applications have *immutable* labels, and $\mathsf{L}_{\text{Img}} \not\subseteq \mathsf{L}_{\text{FC}}$, the enforcement system simply prevents `FormatConvertor` from reading objects with the label L_{Img}. On such systems, labels have to be initialized suitably to allow the desired flows between applications.

Picaros supports *mutable* labels, in which an application can read data after suitable modification of its label. For example, suppose that the label L_{FC} of `FormatConvertor` was initially ϕ. To read data labeled $\mathsf{L}_{\text{Img}} = \{\text{CAMERA:IMAGERAW}\}$, `FormatConvertor` will have to explicitly request data owners whose tags are absent from its label. Prior DIFC systems (*e.g.,* Flume [40]) have enabled such mutable tags by using *capability sets*. In this approach, a data owner, such as `Camera`, permits other downstream applications, such as `FormatConvertor`, to consume its data by adding the tag CAMERA:IMAGE to `FormatConvertor`'s positive capability set. The positive capability set consists of a set of tags that `FormatConvertor` can add to its DIFC label; a similar negative capability set allows removal of tags.

Picaros's approach is largely similar, except that in Picaros, a request from `FormatConvertor` to add a tag is triggered by sending an explicit request to the tag owner (`Camera` owns the tag CAMERA:IMAGERAW) upon a no-reads-up rule violation. Picaros uses ABE to implement DIFC, and capability sets are implemented by suitably providing decryption keys (see Section III). A successful request will result in the addition of CAMERA:IMAGERAW to L_{FC}, following which it can read the data.

Like other DIFC systems, Picaros ensures that labels of data objects published by `FormatConvertor` will inherit tags that are in the application's label, and thus the label of the object published under the ROS2 topic ImgLoRes will also include the tag CAMERA:IMAGERAW. `FormatConvertor` can option-



Boxes denote DIFC labels of corresponding data objects. In this example, we assume that `FormatConvertor` does not attach any of its own tags to the data it produces. We also elide showing application DIFC labels: `FormatConvertor`, `ImgScrubber` and `DNNEncoder` all have the label {CAMERA:IMAGERAW}, thus allowing the flows shown.

Figure 2: Example from Figure 1 adapted for DIFC in Picaros.

ally choose to insert additional tags that it owns to the labels of data objects that it publishes—*e.g.,* it could add the tag FORMATCONVERTOR:IMGLORES to images published under the topic ImgLoRes. With DIFC in force, `Logger` can consume the output of `FormatConvertor` only if `Logger`'s label L_{Logger} satisfies $L_{Logger} \subseteq L_{FC}$. `Logger` can acquire tags in its label with the explicit permission from the corresponding tag creators. Thus, to consume `FormatConvertor`'s output, whose label includes the tag CAMERA:IMAGERAW, `Logger` will also require explicit permission from `Camera`, thereby allowing `Camera` downstream control over its data.

It is well-known that information-flow control systems often become unusable due to the problem of *label creep*, *i.e.,* when data objects accumulate labels to the extent that they become unusable (*e.g.,* [53, 63]). DIFC offers *declassification* as a method to bypass label creep. A declassifier is an application that a data owner entrusts with the privileges to remove some (or all) tags belonging to that data owner from the DIFC label of data objects (*i.e.,* negative capabilities [40]). Removing tags from the data object's label allows that data object to be consumed by more applications. Figure 2 adapts our running example with a declassifier and also shows the DIFC labels of data objects, as implemented on Picaros. In Figure 2, `ImgScrubber` is the application that `Camera` entrusts with data declassification. Its business logic must include functionality to satisfy `Camera`'s domain-specific privacy needs, *e.g.,* `Camera` may require `ImgScrubber` to pixelate people's faces or other identifiers in the image to very low resolution. `Camera` may wish to ensure that downstream applications that externalize data (*e.g.,* `Logger`) cannot consume images before they are processed by `ImgScrubber`. From the perspective of the DIFC system, `Camera` endows only `ImgScrubber` with the privileges to remove the CAMERA:IMAGERAW tag from the image. Any application that consumes data with the CAMERA:IMAGERAW (*e.g.,* `FormatConvertor` and `DNNEncoder`) can only do so with the explicit permission of `Camera`. Note that `ImgScrubber`'s DIFC label $L_{ImgScrubber}$ must still contain the tag CAMERA:IMAGE to allow it to read the output of `Camera`. However, as a declassifier trusted by `Camera`, it is entasked with santizing the data that it publishes and endowed (by `Camera`) with the privilege of publishing data without the tag CAMERA:IMAGE in the DIFC label of the published data.

Observe from Figure 2 that Picaros allows declassifiers such as `ImgScrubber` to both publish and subscribe to the same ROS topic (ImgLoRes). This is an intentional design feature in ROS2, and Picaros leverages it to enable enforcement of DIFC policies without invasive changes to applications. An alternative approach to ensure that `Logger` only consumes scrubbed images would be to modify it to subscribe to a new topic (say, ScrubbedImg) to which only `ImgScrubber` publishes. Indeed, such an approach has been suggested in prior MAC systems [10]. However, implementing this approach requires modifying `Logger`'s code to ensure that it subscribes to ScrubbedImg instead of ImgLoRes, and that `ImgScrubber` is the only application that publishes to ScrubbedImg. The MAC system can ensure the latter properly only in a static setting when the flow graph between applications is known *a priori*. Although `Logger` continues to subscribe to ImgLoRes in Figure 2, Picaros's DIFC enforcement ensures that it cannot directly consume the output of `FormatConvertor`. This is because the label of `FormatConvertor`'s output will include the tag CAMERA:IMAGERAW, and `Camera` will not give `Logger` permission to add this tag to L_{Logger}. In contrast, the output of `ImgScrubber` does not include CAMERA:IMAGERAW, and can thus be consumed by `Logger`.

B. Specifying DIFC policies

In a DIFC system, labels determine how data flows between applications. DIFC systems have historically relied on application developers (or security administrators) to specify how the application's data must be labeled. This process is cumbersome and has hampered wide deployment of DIFC. The key difficulty is that a DIFC label of a data object must meaningfully reflect the kind of data in that object, *e.g.,* its sensitivity level or a descriptive name that suggestive of the type of information in the data object. We observe that ROS2 provides a unique opportunity sidestep this difficulty and ease the label specification problem. In deciding the ROS2 topics to which an application publishes, application developers already invest the work needed to identify the kind of data produced by an application.

Picaros leverages ROS2 topics to suggest default DIFC labels for data objects. When an application such as `Camera` publishes data to a topic such as ImageRaw, Picaros by default attaches a label that includes a tag with the process identifier of the application and the name of the topic. For simplicity of exposition, in this paper, we will use the name of the application instead of the process identifier in the tag, thus resulting in the tag CAMERA:IMAGERAW being included by default in the label of corresponding data object.

While this is the default behavior in Picaros, applications can optionally choose not to attach any additional tags to data objects they create, as we saw in Figure 2, in which `FormatConvertor` chose not to attach any additional tags to the data objects that it publishes. However, if it did choose not to override the default, data that it publishes under the topic ImgLoRes will be labeled {FORMATCONVERTOR:IMGLORES, CAMERA:IMAGERAW} (and an analogous label {FORMATCONVERTOR:IMGHIRES, CAMERA:IMAGERAW} for data published under the topic ImgHiRes). Note that the tag FormatConvertor:ImgHiRes is *not* attached to the data published under the topic ImgLoRes. This labeling scheme has two benefits. First, it intuitively reflects the intent of the application developer for the data created by `FormatConvertor`, and ensures that the label remains consistent with the kind of information in the data object. Second, it prevents unwanted label creep—if the object published under ImgLoRes also had the tag FormatConvertor:ImgHiRes, downstream applications such as `Logger` will require that tag to be able to consume the low-resolution image data. Although `Logger` could potentially explicitly request `FormatConvertor` to endow it with the tag FormatConvertor:ImgHiRes, doing so would unnecessarily give `Logger` the privileges to also access high-resolution images, whose label includes that tag.

C. Enforcing DIFC policies

The goals of the trusted computing base (TCB) that enforces policies in a DIFC system are to: (**G1**) bind labels securely to entities such as data objects; and (**G2**) enforce the no-reads-up rule, *i.e.,* $L_{data} \subseteq L_{reader}$ and no-writes-down rule, *i.e.,*$L_{publisher} \subseteq L_{data}$. The following difficulties make it challenging to achieve these goals in a ROS2 system:

► *Distributed and decentralized nature of ROS2 applications.* ROS2 applications that interact with each other can be spread out over several physical devices. Most prior work on DIFC has focused on non-distributed settings wherein a centralized kernel, such as an operating system [30, 40, 74] or a language runtime [43, 44, 49, 51, 52], forms the TCB that both stores the labels associated with data objects (goal **G1**) and enforces no-reads-up or no-writes-down (goal **G2**). However, a distributed setting such as ROS2 requires the enforcement TCB to span multiple physical devices. The main challenge in this case is to ensure **G1**—when the DIFC system receives a data object together with its label, it must have a way to ensure that the label and the data object are securely bound to each other. That is, it must not be possible for an attacker (*e.g.,* a malicious ROS2 application) to modify the label of the object by adding or removing tags from the label.

DStar [75] applied DIFC to distributed systems, and addressed the challenge above using a trusted OS kernel on each physical device, responsible for enforcing goal **G2** on that platform, and a trusted exporter process per physical device to bind labels to objects before sending them to other platforms. In DStar, exporter processes only talk to their counterpart on the receiving physical devices, which would then bootstrap the DIFC state of the newly-received data object on that device. This establishes a way to securely bind labels to data objects, thus achieving **G1**.

Unfortunately, this approach does not work on ROS2 without invasive changes that fundamentally alter the data path of inter-application communication. After an initial discovery phase to identify matching topics, ROS2 establishes a socket connection, generally over UDP, between a pair of communicating applications for direct communication (see Figure 3). Introducing an exporter process would both break this communication abstraction and introduce additional centralized network elements in the data path (*i.e.,* the exporter processes), which violates the design philosophy of ROS2.

► *Software architecture of ROS2.* ROS2 (and the underlying DDS) is a set of software packages implemented as shared libraries loaded into the application's address space (Figure 3).[‡] Data from the sender is marshaled and sent down the ROS2 software stack, through the socket connection, and to the receiver, where the data is unmarshaled within the ROS2 software stack of the receiver and sent to the application. A pair of applications may send messages under different topics. However, on ROS2, the same socket/port number is used by applications for all messages exchanged, regardless of topic. As a result, aside from establishing the socket connection between a pair of connecting applications, the underlying operating system is largely opaque to details of data exchanged. If the operating system were to enforce no-reads-up (goal **G2**), this lack of visibility would cause it to label the socket (and the receiving process) with a DIFC label that includes all the tags associated with ROS topics transmitted on that socket, in turn leading to label creep. It may be possible to expose more details to the operating system, but this approach would require extensive re-engineering of the entire ROS2 stack and/or applications.

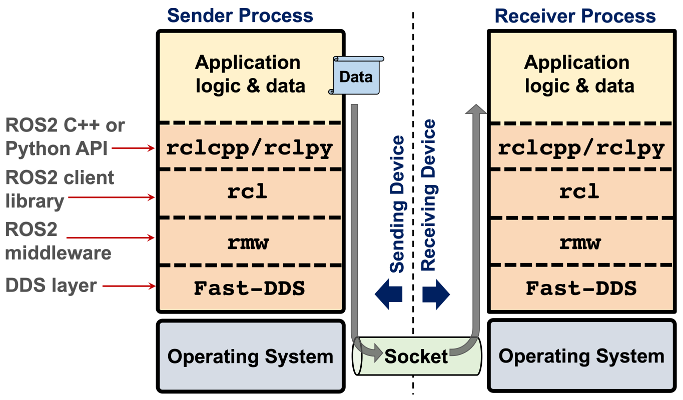


Figure 3: Inter-application communication in ROS2/DDS.

An alternative would be to enhance the ROS2 software stack itself with mechanisms to enforce access control policies. However, this approach poses the question of which application enforces the access control—the sender or the receiver? Both approaches have serious shortcomings:

— *Access control at the receiver:* It is too late to enforce access control at the receiver because the plaintext data will already be in the receiving application's address space, which a malicious application could access via standard exploits. This approach is also poorly suited to the dynamic nature of ROS2 platforms. ROS2 allows applications to start at any time and start publishing/subscribing to messages. Because of the distributed nature of ROS2, an application that starts up on a remote platform may subscribe to messages published under a particular topic. To enforce access control, the publisher must verify that the remote platform enforces receiver-side access control. This is non-trivial to do without attestation hardware on the remote platform, or a centralized trusted entity that can attest the software stack running on the remote platform.

— *Access control at the sender:* If the sender is entrusted with enforcing the access control rule, it must be able to determine the label of the receiving application. In a distributed setting, this would either require centralized state management

[‡]ROS2 does allow multiple applications to execute within a single process, sharing an address space. However, we assume—and Picaros requires (see Section IV)—that each application runs as a separate process.

of DIFC labels or additional queries to a trusted agent on the receiving platform, both of which are unpalatable options. Moreover, such queries must happen *each time* data is transmitted because the label of the receiver may have changed in the interim since the last transmission to the same receiver. It may also lead to time-of-check to time-of-use (TOCTTOU) vulnerabilities if the receiver's DIFC label changes in the interim between the sender's last query of its DIFC state and arrival of the message at the receiver's end.

Motivated by these problems, we sought a fresh approach to DIFC enforcement in Picaros using ABE, as described next.

III. DIFC using ABE

We now provide background on attribute-based encryption (ABE) and discuss how to implement DIFC atop ABE.

A. Background on Attribute-based Encryption

In a cryptosystem based on ABE (originally proposed by Sahai and Waters [12, 61]), plaintext messages are encrypted using an access structure ($\mathscr{A}$) over a set of *attributes*. An attribute is a string that describes some semantic property of the message being encrypted. Intuitively, the access structure $\mathscr{A}$ describes the combination of attributes under which the message is encrypted. The ABE cryptosystem is designed such that an entity can decrypt the message successfully only if it possesses attributes that pass through the access structure $\mathscr{A}$.

For example, suppose that a hospital wishes to use ABE to protect patient data and ensure that the data is accessible only to on-call doctors. The ABE cryptosystem in this case can be defined over a set of attributes that includes Doctor and OnDuty, and patient data is encrypted using the access structure $\mathscr{A}$=Doctor $\wedge$ OnDuty, indicating that only doctors currently on call are authorized to decrypt patient data. Any entity that presents both the attributes Doctor and OnDuty can obtain decryption keys to successfully decrypt the ciphertext. An entity that has only one or none of these attributes will be unable to decrypt patient data.

ABE cryptosystems have evolved over the years to support a variety of expressive access structures under which the plaintext can be encrypted. However, for the purposes of this paper and for all our subsequent discussions, we will restrict $\mathscr{A}$ to a simple Boolean conjunction, *i.e.,* terms connected by AND gates, as in the example above. This restricted version of the access structure suffices to implement DIFC, and is supported by all prior ABE-based cryptosystems.

The central challenge in designing an ABE cryptosystem is to achieve *collusion-resistance*. That is, suppose that an entity has only a subset of the attributes needed to decrypt a piece of ciphertext (*e.g.,* an entity has only the attribute Doctor). Collusion resistance demands that it should not be able to collude with other entities that have the remaining attributes (*e.g.,* an entity that has the attribute OnDuty) to be able to decrypt the ciphertext. Early ABE-based cryptosystems used a trusted centralized authority to generate decryption keys for participants in the system. In centralized ABE schemes, the trusted authority is assumed to know the set of attributes issued to various users of the system. It is entrusted with the task of issuing decryption keys to participants only after checking that the attributes presented by the participant match (or are a subset of) the attributes that are assigned to the participant. The trusted authority then generates a decryption key that is tailored to the set of attributes presented to it. Many ABE cryptosystems proposed to date are implemented using variants of bilinear pairings. We refer the reader to the original paper (*e.g.,* [12]) for details on the cryptographic constructions. We mostly use ABE in a black-box fashion in this paper.

However, the need for a trusted central authority is a key shortcoming of early ABE schemes. This central authority is entrusted with a master key that could be used to produce decryption keys for specific combinations of attributes. Initial attempts were made to eliminate the need for such a central authority by building *multi-authority* ABE schemes [15], which distinguish between the notion of *authorities* and *users*. An authority is any entity that "owns" an attribute, *i.e.,* it can issue keys to decrypt ciphertext that is associated with an access structure $\mathscr{A}$ that has that attribute, while a user of the system is a data consumer that contacts the relevant authorities to obtain these decryption keys. Early multi-authority schemes [15, 16] still required elaborate global coordination between the authorities, but Lewko and Waters [42] developed a fully decentralized multi-authority ABE scheme that avoids the need for any such global coordination after some initial setup. Most current research in the area focuses on variants of such decentralized multi-authority ABE schemes (*e.g.,* [3, 21, 22, 59, 70]). Picaros uses the original construction by Lewko and Waters [42].

Any user that participates in a decentralized multi-authority cryptosystem can obtain the decryption keys by presenting its attributes to the relevant authorities. Users can also encrypt plaintext by defining access structures that use labels created by authorities. Each user in this cryptosystem has a globally-unique identifier (GID). The cryptosystem tailors decryption keys by GID, *i.e.,* a decryption key issued to one user will not work for another user. This feature is the key idea that allows the Lewko-Waters ABE scheme to achieve collusion resistance—two users with different sets of attributes cannot collude to combine their own keys to decrypt ciphertext that each individual is not authorized to decrypt.

Note that in the setup described above, a user of the cryptosystem is not necessarily an authority, *i.e.,* while an authority is one that defines an attribute, any user that is authorized to present that attribute can participate in the system. The idea is that data owners, who wish to define policies on how they data is used, are authorities, while data consumers do not necessarily have to be authorities, and can participate in the system as long as they are assigned a GID. Thus, for instance, in the example discussed earlier, the hospital would be an authority because it defines the attributes Doctor and OnDuty, and an access structure $\mathscr{A}$ over the ciphertext that uses these attributes. However, doctors and other participants in the system simply present the attributes that they have been assigned, and can decrypt the ciphertext if their attributes satisfy the access structure. Formally, the decentralized multi-authority ABE scheme in Picaros has the following API:

① GlobalSetup(λ)$\rightarrow$GP. This API is called once with a security parameter λ to decide the set of publicly-known, global parameters GP of the system that are shared by all participants of this ABE cryptosystem.

② AuthSetup(GP)→(PrivK$_\alpha$, PubK$_\alpha$). Each authority participating in the system invokes this API during setup using the global parameters GP, to output a keypair (PrivK$_\alpha$, PubK$_\alpha$) corresponding to the attribute α that it owns. The authority holds the key PrivK$_\alpha$ privately, and publishes the key PubK$_\alpha$ as its public key. Note that we are using the identifier α to refer to the attribute owned by this particular authority. It must not be confused with the identity of the authority—as mentioned above, each user (and thus authority) also has a globally-unique identifier that will be used in a subsequent API.

The discussion above assumes that each authority owns only one attribute (*i.e.,* a *single-attribute authority*). We have done this to simplify exposition and to keep the API description simple. Practical ABE systems do allow each authority can own more than one attribute, *e.g.,* as in the example discussed above, in which the hospital owns the attributes Doctor and OnDuty. In this case, authority would generate one public/private key pair per attribute that they own. The above AuthSetup API can still be used to simulate applications that require more than one attribute—the application can simply spawn multiple single-attribute authority threads, each of which generates its own public/private key pair using a call to AuthSetup. Thus, in the API call above, the keypair (PrivK$_\alpha$, PubK$_\alpha$) is customized to the single ABE attribute α that this authority owns.

③ Encrypt(M, GP, {PubK$_\alpha$}, $\mathcal{A}$)→C. A user that wishes to encrypt a plaintext message M must provide an access structure $\mathcal{A}$ that determines the attributes (and their logical combination) under which the message must be encrypted. This user must also provide the set of public keys {PubK$_\alpha$} of the corresponding attributes α that appear in the access structure $\mathcal{A}$. In the example above, the hospital is the authority that owns both the attributes Doctor and OnDuty, so the corresponding public key(s) must be provided when a user encrypts any message using an access structure that uses either one of these attributes.

④ Keygen(GID, GP, α, PrivK$_\alpha$)→K$_{(\alpha,GID)}$. When a user with the globally-unique identifier GID wishes to decrypt some ciphertext, this operation proceeds in two phases: the key generation phase (multiple calls by authorities to the Keygen function) and the decryption phase (a call by the user to Decrypt, below). In the key generation phase, the user contacts each authority whose attribute α is used in the creation of the ciphertext (*i.e.,* α is used in the access structure $\mathcal{A}$ used in the corresponding Encrypt API call). It presents its global identifier GID to this authority, which then executes the Keygen API together with its own private key PrivK$_\alpha$ (corresponding to the attribute α) to produce a decryption key K$_{(\alpha,GID)}$ tailored for this particular user and the attribute α.

⑤ Decrypt(C, GP, {K$_{(\alpha,GID)}$})→M. Once the user has obtained the set of decryption keys {K$_{(\alpha,GID)}$} from all the relevant authorities, he invokes this API on the ciphertext C. Note the set of decryption keys {K$_{(\alpha,GID)}$} must be for the same fixed identity GID. The decryption succeeds and produces the plaintext if the set of attributes for which the user presented decryption keys K$_{(\alpha,GID)}$ clears the access structure $\mathcal{A}$ with which the ciphertext was produced; else decryption fails.

We refer readers to the Appendix (and Lewko-Waters [42]) for details of the cryptographic constructions of these APIs. Crucially, the system is fully decentralized except for the one-time global setup that requires the global parameters to be distributed to all participants and the establishment of globally-unique identifiers for all users. Decrypt may require a user to contact the relevant authorities to obtain the decryption key, however, even this communication happens directly between the user and the authorities without any centralized elements.

We now make several observations. First, note that by customizing the decryption key K$_{(\alpha,GID)}$ based on the globally-unique identifier GID of the user requesting the decryption, the ABE scheme ensures that the same decryption key is not provided to two different users. This feature is a critical aspect of the constructions in the Lewko-Waters ABE scheme and is important in providing the collusion-resistance property of the scheme. Second, the decryption key is customized to the attribute α; thus, the same key cannot be used to decrypt a message whose access structure uses a different attribute β. Moreover, only the authority that owns the attribute α can issue a decryption key for that attribute, customized to the user GID requesting that decryption key. Third, the user needs to obtain decryption keys from all relevant authorities whose attributes are needed to pass through the access structure $\mathcal{A}$. In a conjunctive access structure such as used in Picaros, this implies obtaining decryption keys from every authority whose attribute appears in the conjunction. The user cannot decrypt the ciphertext if he has decryption keys for only a subset of the attributes needed to clear the access structure $\mathcal{A}$.

B. Mapping DIFC to Decentralized Multi-Authority ABE

In this section, we describe how DIFC primitives can be mapped to ABE. For this, consider a DIFC system in which a data object $\mathcal{D}$ has a label L$_\mathcal{D}$, and a subject $\mathcal{S}$ (*i.e.,* application that produces or consumes data) has label L$_\mathcal{S}$. By the no-reads-up rule, we have that L$_\mathcal{D}$⊆L$_\mathcal{S}$ if $\mathcal{S}$ has to be able to read $\mathcal{D}$. Moreover, if $\mathcal{S}$ is not a trusted declassifier (*e.g.,* `ImgScrubber` from the example in Section II-A), then by the no-writes-down rule, the DIFC label of any data object $\mathcal{E}$ published by $\mathcal{S}$ must inherit all the tags from L$_\mathcal{S}$, *i.e.,* L$_\mathcal{S}$⊆L$_\mathcal{E}$. This relationship is not necessarily an equality because $\mathcal{S}$ could itself optionally add any additional tags of its choosing to the label of $\mathcal{E}$. However, it cannot remove any tags that are in L$_\mathcal{S}$ when it creates $\mathcal{E}$.

To describe our ABE-based construction of DIFC, we specify how the entities in the DIFC system would be represented in the ABE-based system:

► Every subject $\mathcal{S}$ in the DIFC system is considered as a user in the ABE scheme, is assigned a globally-unique identifier, and is associated with ABE attributes. The attributes of ABE user $\mathcal{S}$ are exactly the set of tags that would appear in the DIFC label L$_\mathcal{S}$. For example, both the `Camera` and `FormatConvertor` applications from Figure 2 would be users in the ABE system that have the attribute Camera:ImageRaw.

► A subject $\mathcal{S}$ in the DIFC system is considered an authority (in the ABE sense) if it owns a tag, *i.e.,* it is the first application to add that tag to the DIFC labels of data objects that it creates. In the example in Figure 2, `Camera` is an authority because it owns the tag Camera:ImageRaw. However, `FormatConvertor` is *not* an authority although by virtue of having Camera:ImageRaw in L$_{FC}$, it is capable of reading data objects that have Camera:ImageRaw in their labels, and also attaches this label to the data objects it publishes.

► A data object $\mathcal{D}$ that has label $\mathsf{L}_{\mathcal{D}}$ in the DIFC system would be represented encrypted with an access structure:

$$\mathcal{A} = \bigwedge_{i=1}^{i=n} \text{TAG}_i, \text{ where } \mathsf{L}_{\mathcal{D}} = \{\text{TAG}_1, \ \ldots, \text{TAG}_n\}$$

$\mathcal{A}$ is thus the Boolean conjunction of all the tags in $\mathsf{L}_{\mathcal{D}}$ considered as attributes. With this construction, observe that data objects with non-empty labels in the DIFC system will always be encrypted in the ABE-based realization as they are passed between subjects. This encryption-based construction ensures that the DIFC label of the object is securely bound to the object's representation, and that it cannot be changed even if the data object moves from one compute platform to another in a distributed system—only a subject that has the corresponding decryption keys will be able to successfully decrypt and read the object.

► A subject $\mathcal{S}$ with the DIFC label $\mathsf{L}_{\mathcal{S}}$ can successfully read a data object $\mathcal{D}$ with label $\mathsf{L}_{\mathcal{D}} \subseteq \mathsf{L}_{\mathcal{S}}$. This is because in the ABE realization, the data object $\mathcal{D}$ will be represented encrypted with an access structure $\mathcal{A}$ that has all the attributes in $\mathsf{L}_{\mathcal{D}}$. $\mathcal{S}$ will posses all the attributes that are used in the access structure $\mathcal{A}$, and can therefore successfully decrypt $\mathcal{D}$. To do so, it would have to obtain decryption keys from the relevant authorities that own the various tags in $\mathsf{L}_{\mathcal{D}}$, which would run the KeyGen function with the globally-unique identifier of the requesting user (*i.e.,* subject $\mathcal{S}$) to produce the decryption key. Observe that $\mathcal{S}$ would have to procure decryption keys only once and cache them locally. For any subsequent data objects received with the same label, $\mathcal{S}$ can simply reuse the corresponding decryption keys directly on the data objects.

A subject that fails to obtain the relevant decryption keys (because it has insufficient attributes) will not be able to obtain clear-text access to the data object. Observe that unlike traditional DIFC, in which the enforcement mechanism determines whether a data object must be released to a requesting subject, in the ABE-based realization, the (encrypted) data object is released to the subject. However, only subjects with the required attributes will be able to procure the keys to successfully decrypt the object.

In this paper, we do not consider support for delegation or revocation, both of which have been studied in prior DIFC-based systems (*e.g.,* Pileus [64]). The original construction of centralized authority ABE [12] provided explicit support for delegation, but we have not explored this possibility in the context of decentralized multi-authority ABE. Although we do not explicitly support revocation, we note that authorities can implement it by simply generating a fresh key-pair, encrypting messages with the new encryption key and re-issuing the corresponding decryption key only to users for whom access privileges are not revoked.

C. Desiderata for ABE-based Realization of DIFC

Given the above construction, we now enlist the desiderata for a secure realization of DIFC using this construction:

D1 *The attributes of a subject $\mathcal{S}$ must be stored securely by the system and must not be available for arbitrary modification by $\mathcal{S}$.* This is because the attributes denote the DIFC label $\mathsf{L}_{\mathcal{S}}$ of the subject. A subject must not be allowed to add to its set of attributes because that would be tantamount to adding tags to its DIFC label. If subjects were allowed to add tags to their DIFC labels, they can arbitrarily increase their privileges to circumvent the no-reads-up rule and read data objects that they are not otherwise allowed to read. Likewise, a subject must not be allowed to delete any of its existing attributes because in a DIFC system data objects published by the subject inherit the DIFC tags of the subject. If the subject were allowed to arbitrarily delete tags, it can declassify and therefore leak data objects. The ability to delete certain DIFC tags from the label of an object must only be given to trusted declassifiers (such as `ImgScrubber` in Figure 2).

D2 *A subject $\mathcal{S}$ must encrypt data objects that it publishes with an access structure $\mathcal{A}$ that is a conjunction of at least all of the subject's attributes.* This requirement encodes the fact that the DIFC label of any data object published by a subject inherits all of the tags in the subject's own DIFC label, *i.e.,* no-writes-down. If the subject in an ABE authority, it can optionally choose to add its own new attributes to the access structure $\mathcal{A}$. Intuitively, this encodes the ability of a subject in a DIFC system to place additional access restrictions on how the data object can be consumed by downstream applications.

In Picaros, both requirements **D1** and **D2** are met by suitably modifying the ROS2 software stack to provide a secure storage area for attributes of applications, and by ensuring that the logic that publishes messages encrypts each data object published with all of the attributes of the publishing application.

D3 *Every application participating in the system must have an unforgeable identity.* Note that ABE already requires each entity to have a globally-unique identifier (GID) that is presented to authorities for the Keygen function. In a practical DIFC system, care must be taken to ensure that the GID presented to the authority indeed corresponds to that of the user, *i.e.,* the system must be able to detect and prevent *impersonation attacks* in which one user impersonates a second user by presenting the second user's GID to the authority, thereby obtaining decryption keys that only the second user is authorized to receive.

Picaros achieves this goal by building upon the existing security primitives of the ROS2+SROS2 stack. It uses traditional public-key infrastructure (PKI) as present in SROS2 to identify users, *i.e.,* as is typical in PKI, public keys represent users, and cannot be forged as long as they are accompanied by digital certificates from a certifying authority. SROS2 already assumes the existence of such a public-key infrastructure (because it uses TLS for inter-application communication), and Picaros simply leverages the same infrastructure to achieve goal **D3**.

In Picaros, applications that are ABE authorities receive requests from other applications to provide a decryption key for each attribute α that they own. For example, the `Camera` application will receive such a request for the attribute CAMERA:IMAGERAW from `FormatConvertor` the first time that `FormatConvertor` receives a message published by `Camera`. Because Picaros builds upon the ROS2 (enhanced with SROS2), this request will be sent over TLS, which allows `Camera` to verify the identity of the requesting application (`FormatConvertor`) and use the corresponding GID when it generates $\mathsf{K}_{(\text{CAMERA:IMAGERAW},\mathsf{GID})}$.

D4 *A secure channel is needed to communicate decryption key $\mathsf{K}_{(\alpha,\mathsf{GID})}$ for an attribute α and user with global identifier*

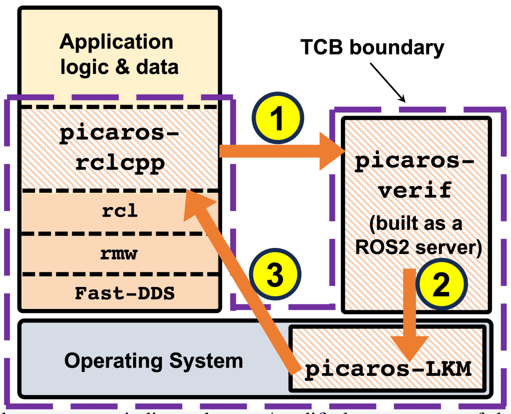
GID to the user from the issuing authority. If a snooping adversary were to get access to the key $\mathsf{K}_{(\alpha,\mathsf{GID})}$, it can decrypt messages that are encrypted with the attribute α. As an example, in Figure 2, if a malicious adversary were to get access to the key $\mathsf{K}_{(\text{CAMERA:IMAGERAW},\text{FC})}$ issued by `Camera` to `FormatConvertor`, the adversary could snoop on and decrypt all messages with DIFC label {CAMERA:IMAGERAW} (*i.e.,* encrypted with $\mathsf{PubK}_{\text{CAMERA:IMAGERAW}}$). Requirement **D4** ensures that decryption keys are received only by intended recipients. As with criterion **D3**, Picaros achieves this goal by leveraging TLS for all inter-application communication. We note that requirement **D4** is orthogonal to the collusion resistance property that Lewko-Waters ABE scheme offers. This property prevents a decryption key for attribute α for one user (with identifier GID) from being used together with the key $\mathsf{K}_{(\beta,\mathsf{GID}')}$ for attribute β for another user with a different identifier GID′, because the Decrypt function requires decryption keys to all be issued to the same GID. However, it does not prevent an eavesdropping user with access to decryption keys issued to a single GID from using those keys to decrypt messages encrypted with the relevant attributes. Requirement **D4** closes this gap.

D5 *Finally, a practical DIFC system must support declassification.* Thus, the system must support a special class of trusted applications that are allowed to decrypt a data object, possibly sanitize the object (*e.g.,* scrub the image in the example of Figure 2), and republish the object with a subset of the tags in its DIFC label. In an ABE-based implementation, the system must allow the data object to be encrypted with a different access structure $\mathcal{A}$ that only has a subset of the declassifier's own attributes, *e.g.,* scrubbed images published by `ImgScrubber` will be published with the access structure $\mathcal{A}$=`True` (or, equivalently, published in the clear). Picaros includes support for trusted declassifiers.

IV. Implementation of Picaros

We have implemented Picaros atop ROS2 release 8 (Humble Hawksbill) with SROS2 patches applied, and eProsima's Fast-RTPS 1.6.0 implementation of DDS. Although ROS2 runs atop various OSes, our prototype targets an Ubuntu 20.04 distribution running the Linux 5.11.0-46 kernel. Figure 4 shows the overall architecture of Picaros, which builds on the ROS2 software stack. Practically, Picaros enforces DIFC policies at the granularity of *SROS2 enclaves* [46, 71–73], which are collections of ROS2 applications that share the same security policy. However, for ease of exposition, it suffices to think of each ROS2 application as an OS-level process, and Picaros as enforcing DIFC policies at the granularity of individual processes. A ROS2 application's process address space consists of the application's business logic, as well as libraries from the ROS2 software stack that are dynamically loaded into the process address space. Picaros has three main components—`Picaros-rclcpp`, `Picaros-LKM` and `Picaros-verif`.

Picaros's **`Picaros-rclcpp`** component enhances `rclcpp` to implement message access control using Lewko-Waters ABE encryption and decryption of messages. Recall from Figure 3 that `rclcpp` is the library that provides ROS2's API to C++ applications.[§] `Picaros-rclcpp` invokes Encrypt on messages sent by the application and Decrypt on messages



Hatched components indicate the new/modified components of the ROS2 software stack that are part of Picaros's TCB. This picture also illustrates how the components interact to modify the DIFC label of the application: `Picaros-rclcpp` forwards the response from the authority to `Picaros-verif` (step 1), which verifies the request and informs `Picaros-LKM` (step 2), which installs the encryption/decryption keys into the state-save area of `Picaros-rclcpp` (step 3).

Figure 4: Components of Picaros on one participating node.

that the application receives over the publish/subscribe channel (`Picaros-rclcpp` currently only supports messages sent via the `rclcpp::SerializedMessage` API). It uses `mcl`'s [48] implementation of the `BN-254` elliptic curve group with the optimal Ate pairing function [8, 68] to implement the Lewko-Waters ABE cryptographic primitives.

Keys used for encryption are based on an application's DIFC labels (*i.e.,* attributes) and must therefore be stored securely without possible modification by the application (criterion **D1** from Section III-C). Picaros achieves this goal by relying on **`Picaros-LKM`**, a loadable kernel module that securely stores artifacts related to the application's DIFC labels, *i.e.,* encryption and decryption keys, in the kernel. However, to ensure that encryption and decryption operations can be implemented in user-space without entailing system calls, Picaros caches these keys in a protected user-space state-save area that `Picaros-rclcpp` allocates during application startup. This state-save area is write- and read-protected from application code by `Picaros-LKM` to ensure that the application cannot directly modify/access the encryption/decryption keys. However, `Picaros-LKM` sets up the area to be readable and modifiable by `Picaros-rclcpp`, which executes in the same address space as the application. Such support can be enabled on modern platforms via *memory domains*, *e.g.,* as implemented in Intel memory protection keys (MPK) [38, 67] or ARM memory domains [6, 17]. `Picaros-LKM` also securely stores the application's globally-unique identifier required by ABE, satisfying criterion **D3** from Section III-C.

The Encrypt code in `Picaros-rclcpp` reads and applies all the encryption keys from this state-save area to encrypt messages published by the application, thus satisfying criterion **D2** from Section III-C. It also attaches metadata to the ciphertext that describes the access structure ($\mathcal{A}$) used to encrypt the message. When a message is received, `Picaros-rclcpp` uses the metadata to locate the corresponding decryption keys to use with the Decrypt code. This operation succeeds in retrieving the plaintext only if the application has all the required decryption keys. We detail how decryption keys are obtained from authorities when we describe `Picaros-verif`.

[§]Picaros currently only supports `rclcpp`. We plan to add support for Python applications by modifying `rclpy` in future work.

API function —	Invoked by —	Invoked when —
GlobalSetup(λ)	Entity that bootstraps the DIFC system.	Once during system bootstrap.
AuthSetup(GP)	Each authority, *i.e.,* owner of a DIFC tag.	Once during setup of each authority.
Encrypt(M, GP, {PubK$_\alpha$}, $\mathscr{A}$)	Any user that wishes to or must attach tag α to messages that it publishes. Invoked by `Picaros-rclcpp` of user.	Once per message packet that is to be published with tag α in its DIFC label.
Keygen(GID, GP, α, PrivK$_\alpha$)	By `Picaros-rclcpp` of authority that owns attribute α, on behalf of user with identifier GID that wishes to obtain their customized decryption key K$_{(\alpha,GID)}$.	Once for each attribute α for each user GID.
Decrypt(C, GP, {K$_{(\alpha,GID)}$})	By `Picaros-rclcpp` of user with identifier GID that wishes to decrypt ciphertext C.	Once for every message packet received and to be consumed by the application.

Figure 5: Summary of how ABE APIs are invoked by various entities in the DIFC ecosystem.

If an application is an authority that owns an attribute, it controls access to how other applications consume data encrypted with that attribute, *e.g.,* `Camera` owns the tag Camera:ImageRaw in the example from Figure 2. `Picaros-rclcpp` implements the `AuthSetup` API of the ABE scheme that is executed when the application starts up, which generates the corresponding public/private key pair(s) for the attribute(s) that the authority owns. The `Camera` application would thus generate PrivK$_{\text{Camera:ImageRaw}}$ and PubK$_{\text{Camera:ImageRaw}}$.

When a downstream application such as `FormatConvertor` first receives a message published by `Camera`, its DIFC label prevents it from consuming the message. As discussed in Section II-A, `FormatConvertor` explicitly requests `Camera` to allow the tag Camera:ImageRaw to be added to its the DIFC label. In Picaros's ABE-based realization of DIFC, this corresponds to `FormatConvertor` lacking the key to decrypt the message sent by `Camera`. The `Picaros-rclcpp` code executing in the context of `FormatConvertor` inspects the attributes in the access structure $\mathscr{A}$ in the metadata of the ciphertext to determine if there are any keys missing to decrypt the message. If missing, it initiates a request to `Camera` to provide the decryption key (in this case corresponding to the attribute Camera:ImageRaw), together with the GID of `FormatConvertor` (obtained from `Picaros-LKM`). This request is communicated over traditional ROS2 channels (enhanced with SROS2), and is therefore TLS-encrypted. The `Camera` application authenticates the identity of the requestor (using traditional TLS mutual authentication) and determines whether to allow `FormatConvertor` to modify its DIFC label. Such determination is application-specific, *e.g.,* only `Camera` can decide which downstream users must be able to read its data. `Picaros-rclcpp` implements KeyGen, which the authority (*i.e.,* `Camera`) uses if it decides to allow the request to generate the corresponding K$_{(\text{Camera:ImageRaw,FC})}$ key. `Camera` sends this key to `FormatConvertor` over TLS, thus satisfying criterion **D4** from Section III-C.

On the requestor's (`FormatConvertor`) side, the component `Picaros-rclcpp` forwards the response received from the authority to the **`Picaros-verif`** component of Picaros. This workflow is illustrated using the arrows in Figure 4. `Picaros-verif` is implemented as a user-space ROS2 server—each physical hardware machine (*e.g.,* a robot) that is part of the distributed robotics platform executes a participating ROS2 application has one `Picaros-verif` component. `Picaros-verif` authenticates the identity of the sender and the digital signature on the message, thus ensuring that was sent by the authority to which the request was sent (*i.e.,* `Camera` in this case). This response contains both the encryption key PubK$_{\text{Camera:ImageRaw}}$ and the decryption key K$_{(\text{Camera:ImageRaw,FC})}$. `Picaros-verif` requests `Picaros-LKM` to add both the encryption and decryption keys into the set of attributes associated with `FormatConvertor`. It does so by saving these keys to the kernel and to the state-save area set up by `Picaros-rclcpp` in `FormatConvertor`'s address space. The decryption key allows `FormatConvertor` to decrypt messages sent by `Camera`, while adding the encryption key is the DIFC equivalent of adding the tag Camera:ImageRaw to `FormatConvertor`'s DIFC label. The latter ensures that all outgoing messages from `FormatConvertor` now have the DIFC tag Camera:Image.

Note that `Picaros-verif` is only invoked *once* per DIFC tag—the first time that a process discovers that it lacks a particular DIFC tag in its label sends a request to the authority that owns that tag. If the request succeeds, and the requestor's DIFC labels are mutated (by adding the relevant ABE keys), then no further requests are necessary. `Picaros-verif` does not appear in the datapath of message exchange after this initial setup, and we therefore do not consider it as a centralized element in the design of Picaros. Observe that (with one exception) all the above support involves *no* changes to the ROS2 application's code, and happens transparently within the components of Picaros. The only exception is for authorities—when an authority receives a request by a downstream application to add a tag to its DIFC label (or become a declassifier), the application must include the code to implement the domain-specific checks needed to determine whether the request must be allowed.

Support for declassifiers in Picaros (criterion **D5** from Section III-C) largely mirrors the workflow that we described above for modifying DIFC labels, with a few notable differences. When it first receives a message from `Camera`, an application such as `ImgScrubber` sends a request to `Camera` to modify its DIFC label. Each such request also includes a Boolean parameter that determines whether the application has also requested declassification privileges (`FormatConvertor` sets this to `False`, while `ImgScrubber` sets it to `True`). At this point, `Camera` must again use domain-specific logic to determine if the request is to be permitted, and if so, the response from `Camera` contains a similar Boolean flag. Once the `Picaros-verif` component for `ImgScrubber` verifies the request, it adds the corresponding decryption key K$_{(\text{Camera:ImageRaw,ImgScrubber})}$ to `ImgScrubber`, but omits the encryption key PubK$_{\text{Camera:ImageRaw}}$. This ensures that while `ImgScrubber` can read messages from `Camera` to which it subscribes, the messages published by `ImgScrubber` are not encrypted by PubK$_{\text{Camera:ImageRaw}}$. This is equivalent to removing the DIFC tag Camera:ImageRaw from the output of `ImgScrubber`, thus in effect declassifying the image.

When the whole system is bootstrapped for the first time, all applications other than authorities have empty DIFC labels. As applications receive messages from authorities, they send requests to add tags to their DIFC labels, *i.e.,* encryption/decryption keys corresponding to various attributes. Figure 5 summarizes how the ABE API is invoked by various components of Picaros to implement DIFC.

A. Trusted Code Base

As is the case with other DIFC systems, all the core components that implement DIFC enforcement logic are part of Picaros's TCB. In this case, `Picaros-rclcpp`, `Picaros-LKM`, `Picaros-verif`, the OS and all supporting libraries are in the TCB. Note that even SROS2's security guarantees require the entire ROS2 stack (starting downward from `rclcpp`, to the OS) to be part of the TCB, and Picaros is similar in that respect.

Picaros has both kernel-space and user-space components in the TCB. We assume that a robotics platform employs standard hardware-based attestation methods at boot time to ensure that the code of the OS kernel and `Picaros-LKM` have not been tampered with. The kernel must provide standard process-level isolation and protection mechanisms (*e.g.,* data-execution prevention) to protect `Picaros-verif` and `Picaros-rclcpp` from several classes of runtime attacks. In particular, data-execution prevention ensures that a malicious application cannot use runtime exploits to execute new attacker-provided code in place of the mechanisms in `Picaros-rclcpp`. As discussed earlier, `Picaros-rclcpp`'s state-save area is allocated in the address-space of the (untrusted, possibly malicious) application. `Picaros-LKM` writes data structures storing encryption and decryption keys to this state-save area, and these must be readable by `Picaros-rclcpp`, but not by the application code. This can be realized with hardware support in the form of memory domains (either using Intel MPK [38, 67] or ARM memory domains [6, 17]) to provide such support. Even if such hardware support is lacking, read- and write-protection of the state-save area can be implemented using binary rewriting of the application's executable code to enforce software-fault isolation [69], which ensures that read and write operations from the application's code do not target the state-save area.

Although Picaros assumes that defenses such as data-execution prevention and memory domains are in place, they still admit certain attacks that we consider orthogonal to the contributions of this paper. These include runtime exploits that can subvert the integrity of the OS kernel and `Picaros-LKM`, or memory-error attacks in the C++ application such as control-flow hijacking and return-oriented exploits. Standard code-hardening methods (*e.g.,* control-flow integrity [1]) can be used to increase the application's and kernel's robustness against these attacks. Note that applications in ROS2 (with SROS2) are also subject to the same threats, and that the security guarantees provided by ROS2 can also be subverted using these attacks. Picaros's goal is simply to provide information-flow control over and above the security features enabled by SROS2, and therefore we consider these attacks out-of-scope.

B. Threat Analysis

We now discuss Picaros's defenses against unauthorized label modification and collusion attacks.

► *Unauthorized DIFC label modification.* An application that is able to maliciously modify its DIFC label can read data objects that it is not authorized to, by adding tags to its DIFC label. It can similarly leak information by removing tags of other authorities from its DIFC label, which then ensures that the DIFC label of messages that it publishes lack this tag.

Picaros's components prevent unauthorized modification of ABE artifacts that denote DIFC labels. Any tag additions to the DIFC label correspond to adding the encryption key PubK_α of the corresponding attribute α. These keys are publicly available from the corresponding authorities. However, in Picaros, any requests to add encryption keys must be initiated by the corresponding authority. This request must flow through `Picaros-verif`, which checks the authenticity of the message's source and its contents (using TLS), before invoking `Picaros-LKM` to add this encryption key to the application's key store, both in the kernel and in the state-save area in `Picaros-rclcpp`. Likewise, any requests to add decryption keys such as $\mathsf{K}_{(\alpha,\mathsf{GID})}$ to the application's keystore are also authenticated by `Picaros-verif` to check the source of the request. Any requests to delete of public keys (particularly by declassifiers) are handled similarly, and must be approved by an authority and verified by `Picaros-verif`. The untrusted application itself does not have access to either the in-kernel keystore or the isolated state-save area (protected by memory domains), and therefore cannot add or delete keys without the explicit approval of `Picaros-verif`.

► *Collusion attacks.* The Lewko-Waters ABE construction prevents an important class of collusion attacks, in which one user that has decryption keys for attribute α colludes with another user that has decryption keys for attribute β to obtain the ability to decrypt a message that has both attributes. It prevents this attack by customizing decryption keys based on the user's globally-unique identifier (GID). In Picaros, `Picaros-rclcpp` sends Keygen requests to the authority over TLS, with the GID retrieved from `Picaros-LKM`. Authorities verify that the source of this request (as determined by TLS) matches the GID of the requestor, and only then send the customized decryption key to `Picaros-rclcpp`, which is then added to the requestor's keystore via `Picaros-verify`.

While Lewko-Waters prevents attacks that involve combining decryption keys with different GIDs, two users can still collude in other ways to bypass DIFC. For example, consider a malicious application `BadFC` that that also subscribes to the ROS topic Camera:ImageRaw. `Camera` may wish to share data only with `FormatConvertor` but not with `BadFC`, and thus reject any requests from `BadFC` to add the tag CAMERA:IMAGERAW to its DIFC label. Practically, in Picaros, this means that any attempts by `BadFC` to obtain the decryption key $\mathsf{K}_{(\text{CAMERA:IMAGERAW},\mathsf{BadFC})}$ will fail. However, `FormatConvertor` can obtain the decryption key $\mathsf{K}_{(\text{CAMERA:IMAGERAW},\mathsf{FC})}$ legitimately from `Camera`, and collude with `BadFC` by simply this share decryption key. This key does not have to be part of `BadFC`'s keystore—indeed, `Picaros-verif` will reject any attempts by `BadFC` to add this decryption key to the keystore because the request will fail source authentication. Instead, because `BadFC` has subscribed to the topic Camera:ImageRaw, it will continue to receive (encrypted) messages published to this topic. `BadFC` can log these messages and decrypt them offline with the key $\mathsf{K}_{(\text{CAMERA:IMAGERAW},\mathsf{FC})}$. The $\mathsf{Decrypt}$ function in the Lewko-Waters ABE scheme only requires multiple decryption keys used in the same request to share the same GID, which this (single) decryption key trivially satisfies. Criterion **D4** in our desiderata in Section III-C was motivated by this observation.

Picaros prevents such collusion attacks by read-protecting decryption keys from untrusted application code. When `Camera` sends $\mathsf{K}_{(\text{Camera:ImageRaw,FC})}$ to `FormatConvertor`, the key is stored in `Picaros-LKM`'s keystore and cached in `Picaros-rclcpp`'s state-save area, both of which are read-protected from the business logic of the `FormatConvertor` application.

V. Evaluation

The primary goal of our evaluation was to understand the performance of Picaros's ABE-based implementation of DIFC. Our implementation of Lewko-Waters ABE scheme uses bilinear pairings (optimal Ate pairings) from the `BN-254` elliptic curve group [8, 68]. Recall from the discussion in Section III-A that both the Encrypt (and Decrypt) functions take a *set* of keys, depending on the number of attributes to encrypt (or decrypt) the message with. The implementation of these functions in the Lewko-Waters scheme involves modular exponentiation operations. The number of exponentiation operations is proportional to the number of keys used in the operation, *i.e.,* attributes associated with the message. This in turn depends on the number of tags in the message's DIFC label. Our evaluation measures the cost of cryptographic operations as a function of DIFC label size (Section V-A), the latency of ABE-based DIFC in microbenchmark ROS2 pipelines (Section V-B), and end-to-end performance of Picaros on ROS2 application benchmarks (Section V-C).

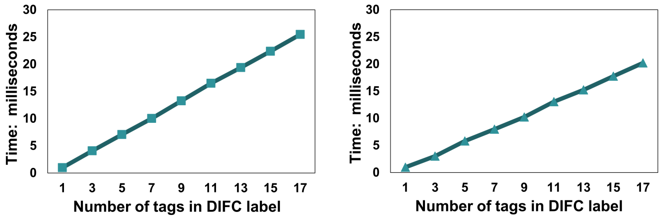
All measurements reported in this section use ROS2 benchmarks in which data publishers continuously produce new messages every 100 milliseconds (10Hz), that are sent to subscribers. Each experiment consists of running a benchmark for a duration of 60 seconds and collecting the relevant measurements (*e.g.,* encryption/decryption cost, latency or resource consumption) observed during that run of the benchmark. Our measurements are averaged over five such experiments.

We measured the performance of encryption/decryption and application latency on a 8-core Intel Corei7-7700 CPU (3.60GHz) with 32GB RAM, running Ubuntu 20.04 (Linux kernel 5.11.0). By default, the ROS2/DDS stack do not guarantee that a message sent by a publisher will be received by a subscriber. Rather, it exposes several QoS configuration parameters that can be tuned based on site-specific requirements. We used the `reliable` QoS policy with queue depth of 10, which we observed minimized message loss in our experiments.

A. Encryption and Decryption Performance

To reliably measure the performance of encryption/decryption, we created a two-node ROS2 application pipeline, with one publisher and one subscriber. The publisher is the authority that adds tags to the DIFC label of the message it publishes (*i.e.,* encrypts) based on the topic(s) published, and the subscriber is given the required tags in its DIFC label to allow it to successfully decrypt the message.

Figure 6 reports the results of our experiments. We measured the performance of encryption at the publisher and decryption at the subscriber by varying the number of DIFC tags in the label. We fixed the message size at 128-bytes for this experiment; we also conducted this experiment with 8-byte and 2048-bytes, and the performance numbers (not reported here) were comparable. The raw cost of encryption and decryption is on the order of a few milliseconds, and increases linearly with the number of tags in the DIFC label. This is because each additional tag corresponds to additional modular exponentiation operations with the corresponding encryption/decryption key.



6(a) Encryption time **6(b) Decryption time**

Figure 6: Performance of ABE encryption and decryption.

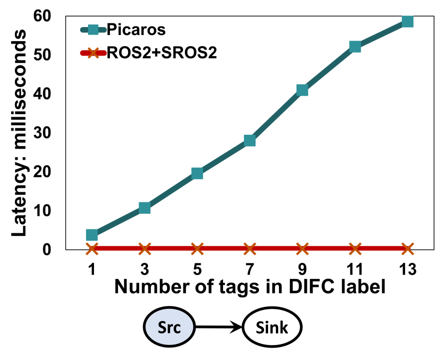
B. Microbenchmark Latency

While Figure 6 reports the raw cost of ABE cryptographic operations, it is also important to quantify the impact of these operations on in terms of end-to-end latency. We measure latency as the time elapsed between the publication of a message at a source node (publisher) to its delivery at a sink node (subscriber) in various ROS2 application pipelines. We compare the latency of these pipelines running on Picaros with their performance on an SROS2 platform, which is our baseline. We conducted experiments with three pipelines:

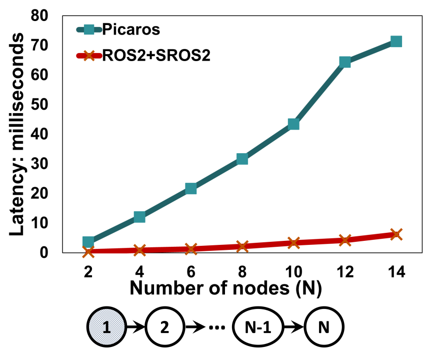
① *Single-source single-sink.* We consider a two-node application pipeline with one node as the publisher (and authority), while the second node is the subscriber (identical to the one in Section V-A). We vary the number of topics to which the authority publishes, which in turn impacts the number of tags in its DIFC label. The message size is 128-bytes for all our latency experiments.

Figure 7(a) shows the result of our latency measurements. The latency observed on SROS2 varied between 250-300μs in our experiments as we increased the number of topics to which the authority publishes. On Picaros, the latency for the same benchmark varies from 3ms to 59ms as the number of topics is increased. We observed that the increase in latency is roughly linearly proportional to the number of tags in the DIFC label, a trend also illustrated in Figure 6. Note that the raw latency jumps from a few hundred microseconds on SROS2 to tens of milliseconds on Picaros. While we attribute this increase in part to bilinear pairing operations of ABE, we also note that the `mcl` library that we use for these operations is far less mature and less optimized than the cryptographic operations implemented in the TLS library used by SROS2.

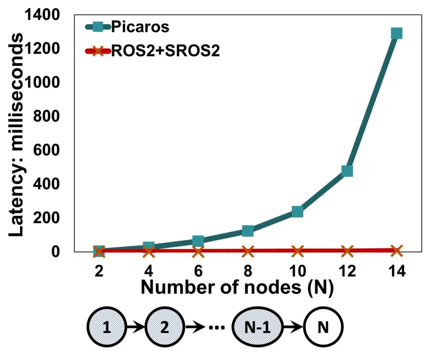
② *N-node chain, single-authority.* For this experiment, we considered a straight-line topology with *N* nodes, as illustrated in Figure 7(b). The first node in this chain is the sole authority (shown as the shaded node)—it publishes to a single topic, and the tag corresponding to this topic is added to the DIFC label of the message it publishes. The other nodes simply forward this message to the next node but republish it under their own topics. That is, the node decrypts the message and re-encrypts the message with a single key corresponding to that of the authority's topic/DIFC tag. However, intermediate nodes are not authorities and do not add any further tags to the DIFC label. The message reaches the last (sink) node with a single



7(a) Single-source single-sink



7(b) N-node chain with single authority



7(c) N-node chain with N-1 authorities

Figure 7: Latency with single-node single-sink and N-node chain topologies.

tag in the DIFC label, and we measure the latency to receive the message at the sink by varying N.

③ *N-node chain, N-1 authorities.* This experiment considers a topology similar to the previous one, but each of the $N-1$ nodes on the path to the sink node is also an authority, as shown in Figure 7(c). Thus, as each intermediate node decrypts republishes the message under its own topic, it adds its own tag (one each) to the DIFC label of the message. The message keeps accumulating tags in its DIFC label and reaches the sink with $N-1$ tags in its DIFC label.

Figure 7(b) and 7(c) report the results of these experiments. The latency on SROS2 shows a roughly linear trend. In Figure 7(b), the latency increases roughly linearly with N because each node in the chain performs one decryption and re-encryption operation. In contrast, the latency with increasing values of N in Figure 7(c) mirrors the fact that the number of ABE cryptographic operations increases quadratically with N. This is because the message reaching the i^{th} node has $(i-1)$ tags. This message must be decrypted with $(i-1)$ keys, and reencrypted with i keys, including that of the the i^{th} node's own tag. This is DIFC's default label creep behavior with $N-1$ authorities, and without declassifiers, and represents the "worst case" application pipeline for Picaros.

C. End-to-end Benchmark Performance

To measure the end-to-end latency and resource consumption (memory usage and power consumption) of Picaros on real ROS2 applications, we use benchmarks from the iRobot suite [2]. As before, we compare the performance against the SROS2 platform as the baseline. iRobot has three application pipelines—`Cedar`, `Sierra Nevada`, and `Mont Blanc`. `Sierra Nevada` and `Cedar` are application pipelines with 10-nodes, while `Mont Blanc` is a 20-node topology. The nodes are connected in complex topologies with each node publishing and subscribing to a number of topics. There are multiple paths (including loops) from a data publisher to a consumer in these topologies. Each node of the application pipeline runs as a separate process on the underlying platform, and we run the entire pipeline on a single hardware platform.

In each of `Sierra Nevada` and `Cedar`, we designate a single node as the sole authority that adds a single tag to the DIFC label of messages it publishes. For `Mont Blanc`, we repeat the experiment with three configurations, in which there are one, four and seven nodes that we designate as authorities, each adding a single tag to the DIFC label of messages they publish.

Topology ↓	Path length	Latency (in milliseconds)	
		SROS2	Picaros
`Cedar`	3	0.85	10.4
`SierraNevada`	3	0.94	13.6
`Mont Blanc-1`	5	1.34	61.3
`Mont Blanc-4`	5	1.34	115.0
`Mont Blanc-7`	5	1.34	316.9

Figure 8: Latency experiments with iRobot benchmarks.

Topology ↓	Memory usage (MBs)		Power draw (mWatts)	
	SROS2	Picaros	SROS2	Picaros
`Cedar`	1690.1	2525.1 (+49.4%)	4896.7	5437.0 (+11.0%)
`SierraNevada`	2163.1	2442.5 (+12.9%)	4881.0	5393.0 (+10.5%)
`Mont Blanc-1`	2529.3	4019.6 (+58.9%)	5056.1	5281.8 (+4.5%)
`Mont Blanc-4`	2529.3	4068.2 (+60.8%)	5056.1	5295.7 (+4.7%)
`Mont Blanc-7`	2529.3	4096.8 (+61.9%)	5056.1	5307.7 (+4.9%)

Figure 9: Resource consumption with iRobot benchmarks.

Because iRobot benchmarks have complex topologies that often have multiple paths between two nodes, we measured the latency for a single data packet to traverse from a designated source node to a sink node via the shortest path. Figure 8 presents the results of our latency measurements; it also shows the length of the shortest path from the source to the sink. Note that in `Mont Blanc-4` and `Mont Blanc-7` multiple nodes are authorities, whereas in the others, there is only one authority. Nevertheless, in all benchmarks, we assume that the authority(ies) permit all subscribers to read the data—thus intermediate nodes along a path also decrypt and re-encrypt packets, as explained for the chain topologies discussed earlier. We observe that the latency increases from about a millisecond on SROS2 to a few tens/hundreds of milliseconds on Picaros. This is expected due to the cost of decrypting packets at intermediate nodes that subscribe to the topic, and re-encrypt packets with the authority's label. We also observe (via various `Mont Blanc` configurations) that the latency increases as the number of authorities increases. This is because the intermediate nodes must decrypt/re-encrypt with keys corresponding to every tag in a message's DIFC label.

Figure 9 presents the results of the resource-consumption experiments. For these experiments, we used an Nvidia Xavier NX development board, that has a 6-core Nvidia Carmel ARMv8.2 64-bit CPU and 8GB RAM. This board has built-in power-measurement hardware that allows us to record the power-draw on the system 5V rail. The memory utilization overhead of Picaros over SROS2 varies from 12.9% for `Sierra Nevada` to 61.9% for `Mont Blanc-7`, while the power-draw overhead varies from 4.5% for `Mont Blanc-1` to 11.0% for `Cedar`. Observe that the SROS2 numbers for all three `Mont Blanc` topologies are the same because the notion of authorities only applies to Picaros.

VI. Security Evaluation: A Case Study

We illustrate the end-to-end functionality of Picaros using a real-life ROS2 application pipeline taken from NVidia's Isaac ROS platform demos (Figure 10). Here, `RealsenseCamera` is a ROS2 node that publishes raw images to the topic camera/color/image_raw (aliased to T1 in the figure) which are converted to tensors via a pair of applications (`isaac_ros_dnn_encoder` and `isaac_ros_tensor_rt`). These tensors are published to the topic tensor_sub, and used by downstream ROS2 nodes. In particular:

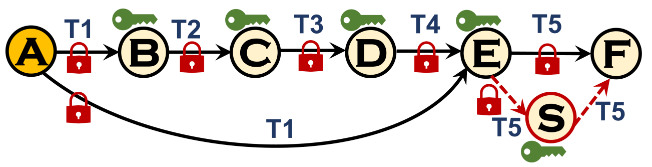
► `YoloV5DecoderNode` performs object detection with the detected object boundaries published to object_detections, and

► `VisualizationNode`, which subscribes to both the topics object_detections and camera/color/image_raw and publishes an image overlaid with object boundaries under the topic yolo5_processed_image.

Now suppose that the owner of the `RealsenseCamera` application wishes to exert control over which downstream applications have access to images (either in the raw or processed form) derived from the `RealsenseCamera`. Without DIFC, it is hard to exert such control, and downstream applications such as the monitoring tool `Foxglove`, which subscribes to the messages published by `VisualizationNode`, can obtain images that `RealsenseCamera` produces.

With DIFC as implemented in Picaros, we designate the `RealsenseCamera` application as an authority (depicted as the shaded circle Ⓐ in the figure). All messages published by `RealsenseCamera` will have the DIFC label {RealsenseCamera:camera/color/image_raw}—we abbreviate this label as {RscImg} for the rest of this discussion. Observe that this DIFC label is automatically derived from the name of the publishing application and topic name of the messages to be protected. In Picaros, the task of attaching a DIFC label is realized by encrypting the published messages with the key $\mathsf{PubK}_{\text{RscImg}}$, and in Figure 10, this is depicted with a lock icon on the outgoing message. `RealsenseCamera` must explicitly allow each downstream application (with unique global identifier GID) to consume this message by decrypting it using the key $\mathsf{K}_{(\text{RscImg},\text{GID})}$ (shown with the key icons in Figure 10). For this case study, we assume that `RealsenseCamera` grants all intermediate applications (except `FoxGlove`) permissions to consume messages labeled with {RscImg}. Every intermediate application in the pipeline that reads messages with the label {RscImg} also preserves this label in the message that it publishes, *i.e.,* their outgoing messages must be encrypted with the key $\mathsf{PubK}_{\text{RscImg}}$, shown again with the lock icon in Figure 10.

`RealsenseCamera` may wish to prevent network-facing monitoring applications such as `FoxGlove` from accessing sensitive data contained the images. While it can simply prevent access by refusing to generate the decryption key $\mathsf{K}_{(\text{RscImg},\texttt{FoxGlove})}$ for `FoxGlove`, this may preclude some useful applications from functioning properly. Thus, `RealsenseCamera` can designate a sanitizing app, in this case `Sanitizer`, to suitably modify the image so that all sensitive content is removed. `Sanitizer` subscribes to the topic yolo5_processed_image and `RealsenseCamera` gives it the decryption key $\mathsf{K}_{(\text{RscImg},\texttt{Sanitizer})}$ to process the image and publish it under the same topic. However, `Sanitizer` publishes its output in the clear under the topic yolo5_processed_image. Observe that although `FoxGlove` subscribes to this topic and receives messages from both `Sanitizer` and `VisualizationNode`, only the sanitized messages from `Sanitizer` are cleartext images.



Legend:

	Ⓐ: `RealsenseCamera` (Authority)
	Ⓑ: `isaac_ros_dnn_encoders`
T1: camera/color/image_raw	Ⓒ: `isaac_ros_tensor_rt`
T2: tensor_pub	Ⓓ: `Yolov5DecoderNode`
T3: tensor_sub	Ⓔ: `VisualizationNode`
T4: object_detections	Ⓕ: `FoxGlove`
T5: yolo5_processed_image	Ⓢ: `Sanitizer`

Figure 10: Application pipeline used in the case study.

VII. Related Work

ROS Security. We provide general background on ROS security, without specifically referring to ROS1 (ROS version 1) or ROS2. Early studies on ROS security focused on the ROS1 platform [47], but newer work generally targets the ROS2 platform, as ROS1's architecture has known scalability issues and is now deprecated. ROS by itself does not provide security, the community has proposed several techniques to add security mechanisms. These efforts include adding mandatory access control (MAC) policy enforcement to ROS [5, 10], improving messaging secrecy and integrity [14, 28, 56, 66], techniques to secure the underlying DDS layer (specific to ROS2) [45, 73], and other methods to improve ROS application security [25, 26, 37, 72]. SROS2 [27, 46, 62, 71] represents an effort to upstream some of these methods to the ROS2 stack, and is built primarily atop the API of DDS, with patches to the ROS2 as well. It makes TLS standard for all application communication and ensures that publish/subscribe messaging adheres to advertised manifests, thereby eliminating several common classes of attacks such as eavesdropping, message spoofing and corruption. Recent work on model checking SROS2 has uncovered security holes in SROS2 itself [24].

Picaros builds on the basic security guarantees provided by ROS2/DDS enhanced with SROS2. While the bulk of prior work on access control for ROS2 applications has focused on providing MAC policy enforcement [4, 5, 10], the issue of information-flow control and providing applications downstream control over their data remains open. To our knowledge, Picaros is the first to address this problem using DIFC.

Decentralized Information-Flow Control. Information-flow control dates back to the classic papers by Bell-LaPadula [11] and Biba [13]. However, in these systems labels on subjects and data objects were assigned centrally by a system administrator. DIFC, first introduced by Myers and Liskov [50], makes label assignment egalitarian, allowing subjects to add their own tags to the DIFC labels of data objects they own. DIFC concepts have since been applied to several language-based systems (*e.g.,* [43, 44, 49, 51]), operating systems (*e.g.,* [30, 40, 74]), web services [18, 19] and mobile systems [53]. All these systems assume that a developer or end-user defines the DIFC labels for an application.

Most of these DIFC systems were tailored for individual machines with a centralized policy-enforcing TCB, which is either the language runtime, OS kernel, or middleware that sits below the application. DStar [75] first generalized DIFC to distributed systems, with each participating machine running a HiStar [74] DIFC kernel and a dedicated process to bind DIFC labels to data objects during export/import from other machines. Picaros embraces decentralization as a central design principle and avoids any centralized elements in its design, and is the first DIFC system for ROS2. It also eases DIFC label assignment for applications by deriving labels using the ROS2 topics from the application's manifest. To our knowledge, Picaros is also unique in using ABE as the core mechanism for DIFC policy enforcement.

Cryptographic Schemes for Information-Flow Control. Although several cryptographic solutions, including ABE [12, 61] and many variants (*e.g.,* [9, 34, 41]), have been developed for access control using attributes (including user identity), Damgard *et al.* [20] were the first to develop cryptographic support for *information flow* as classically defined by Bell-LaPadula [11] and Biba [13]. The primary focus of Damgard *et al.*'s work is to develop new cryptographic protocols for various elements of information flow control, including declassification. This work was followed up by several others [7, 33, 36, 39, 65] that improved upon their protocols, including the work of Han *et al.* [36], who modeled the problem of information-flow control using ABE primitives.

Picaros differs from these prior works in several ways. First, Picaros focuses on DIFC, in which individual users can define DIFC tags to add to the data objects they produce, whereas prior work considers traditional, Bell-LaPadula-style centralized information-flow control. Second, Picaros is tailored for the decentralized environment of ROS2 and therefore uses decentralized multi-authority ABE as the core cryptographic primitive. In contrast, the cryptographic primitives developed in prior work, including the ABE-based work of Han *et al.*, use centralized elements in their solutions, *e.g.,* to model declassification using a centralized server called a sanitizer. Last, while prior work has primarily focused on developing new protocols, to our knowledge, Picaros is the first practical implementation applied to enforce DIFC on a real-world system, namely ROS2. Although subsequent work has improved upon the Lewko-Waters ABE scheme in multiple ways (*e.g.,* [21, 22, 65]), the expressivity of the Lewko-Waters scheme proved sufficient for Picaros.

VIII. Summary

As robotics platforms that use ROS2 gain in popularity, there will be an increasing need to offer applications on these platforms security primitives to safeguard their data. To date, the focus of the ROS2 community has largely been to provide message encryption, authentication and mandatory access control. This paper argues that information-flow control is an essential primitive for ROS2 platforms, and that methods from prior DIFC systems cannot directly be used on ROS2. It presents Picaros, which casts the DIFC problem within the framework of ABE, and shows that this solution ideally suits the decentralized and distributed nature of ROS2 platforms.

Current Affiliations and Acknowledgments: This work was completed when all authors were affiliated with the Indian Institute of Science. Nishit V. Pandya is currently with the University of California–San Diego, Himanshu Kumar with Wells Fargo India Development Centre, and Gokulnath M. Pillai with Cerebras Systems India. Effective mid-2024 their contact information will change to: `{nishitv,himanshukuma,gokulnathp}@alum.iisc.ac.in`. Grants from the Department of Science and Technology's National Mission on Inter-Disciplinary Cyber-Physical Systems (via IHUB-IIT Kanpur), and the National Security Council (via Indian Urban Data Exchange) funded our work. We are grateful to Earlence Fernandes and the NDSS'24 reviewers for their comments on earlier drafts of this paper.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, 2005.
[2] E. Ackerman. iRobot Launches Create-3, with ROS2 Built In. *IEEE Spectrum*, April 2022.
[3] M. Ambrona and R. Gay. Multi-authority ABE, revisited. In *IACR Cryptology ePrint Archive, Report 2021/1381*, 2021.
[4] AppArmor–an effective and easy-to-use Linux application security system. https://gitlab.com/apparmor/apparmor/wikis/home/.
[5] AppArmor and ROS. http://wiki.ros.org/SROS/Tutorials/AppArmorAndROS.
[6] ARM Developer Suite Developer Guide, Version 1.2. Section 7.5.2: Memory access permissions and domains. https://developer.arm.com/documentation/dui0056/d/caches-and-tightly-coupled-memories/memory-management-units/memory-access-permissions-and-domains.
[7] C. Badertscher, C. Matt, and U. Mauer. Strengthening Access Control Encryption. In *Adv. in Cryptology—Asiacrypt*, 2017.
[8] P. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In *International Workshop on Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, 2005.
[9] A. Barth, D. Boneh, and B. Waters. Privacy in encrypted content distribution using private broadcast encryption. In *Financial Cryptography and Data Security*, 2006.
[10] R. Beck, A. Vijeev, and V. Ganapathy. Privaros: A Framework for Privacy-Compliant Delivery Drones. In *ACM Conference on Computer and Communications Security*, 2020.
[11] D. E. Bell and L. LaPadula. Secure computer system: Unified exposition and Multics interpretation, March 1976. Tech. Report MTR-2997, MITRE Corporation.
[12] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *IEEE Symposium on Security and Privacy*, 2007.
[13] K. J. Biba. Integrity considerations for secure computer systems, June 1975. Technical Report MTR-3153, MITRE Corporation.
[14] B. Breiling, B. Dieber, and P. Schartner. Secure communication for the robot operating system. In *Annual IEEE International Systems Conference*, 2017.
[15] M. Chase. Multi-authority Attribute-based Encryption. In *Theory of Cryptography Conference*, 2007.
[16] M. Chase and S. Chow. Improving privacy and security in multi-authority attribute-based encryption. In *ACM Conference on Computer and Communications Security*, 2009.
[17] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. Shreds: Fine-grained Execution Units with Private Memory. In *IEEE Symposium on Security and Privacy*, 2016.
[18] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *ACM Symposium on Operating Systems Principles*, 2007.

[19] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *USENIX Security Symposium*, 2007.

[20] I. Damgård, H. Haagh, and C. Orlandi. Access control encryption: Enforcing information flow with cryptography. In *Theory of Cryptography Conference*, 2016.

[21] P. Datta, I. Komargodski, and B. Waters. Decentralized Multi-authority ABE for DNFs from LWE. In *Advances in Cryptology—Eurocrypt*, 2021.

[22] P. Datta, I. Komargodski, and B. Waters. Decentralized Multi-authority ABE for sfNC^1 from BDH. *Journal of Cryptology*, 36(2), 2023.

[23] Data Distribution Service (DDS). https://www.omg.org/spec/DDS/1.4/PDF.

[24] G. Deng, G. Xu, Y. Zhou, T. Zhang, and Y. Liu. On the (in)security of Secure ROS2. In *ACM Conference on Computer and Communications Security*, 2022.

[25] B. Dieber, B. Breiling, S. Taurer, S. Kacianka, S. Rass, and P. Schartner. Security for the Robot Operating System. *Robotics and Autonomous Systems*, 98, 2017.

[26] B. Dieber, S. Kacianka, S. Rass, and P. Schartner. Application-level security for ROS-based applications. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2016.

[27] B. Dieber, R. White, S. Taurer, B. Breiling, G. Caiazza, A. Cortesi, and H. Christensen. Penetration testing ROS. *Robot Operating System: The Complete Reference (Volume 4)*, 2019.

[28] R. Doczi, F. Kis, B. Suto, V. Poser, G. Kronreif, E. Josvai, and M. Kozlovszky. Increasing ROS 1.x communication security for medical surgery robot. In *IEEE International Conference on Systems, Man, and Cybernetics*, 2016.

[29] T. Dull. Open-Source Robot Operating System and AWS RoboMaker, November 2018. https://aws.amazon.com/blogs/opensource/open-source-robot-operating-system-ros-aws-robomaker.

[30] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Maziéres, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *ACM Symposium on Operating Systems Principles*, 2005.

[31] A. Faz-Hernandez, S. Scott, N. Sullivan, R. S. Wahby, and C. A. Wood. Hashing to Elliptic Curves. Internet-draft, Internet Engineering Task Force, June 2022. https://datatracker.ietf.org/doc/draft-irtf-cfrg-hash-to-curve/16/.

[32] Organization for the Advancement of Structured Information Standards (OASIS) Standard. MQTT version 5.0, March 2019. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[33] G. Fuchsbauer, R. Gay, L. Kowalcyk, and C. Orlandi. Access Control Encryption for Equality, Comparison, and More. In *Public-Key Cryptography–PKC*, 2017.

[34] V. Goyal, A. Jain, O. Pandey, and A. Sahai. Bounded ciphertext policy attribute based encryption. In *International Colloquium on Automata, Languages and Programming*, 2008.

[35] Object Management Group. About the Data Distribution Service Specification Version 1.4. https://www.omg.org/spec/DDS/About-DDS/.

[36] J. Han, L. Chen, W. Susilo, X. Huang, A. Castiglione, and K. Liang. Fine-grained information-flow control using attributes. *Information Sciences*, 484, May 2019.

[37] J. Huang, C. Erdogan, Y. Zhang, B. Moore, Q. Luo, A. Sundaresan, and G. Rosu. RosRV: Runtime verification for robots. In *International Conference on Runtime Verification*, 2014.

[38] Intel. Intel-64 and IA-32 architectures software developer's manual, 2018.

[39] S. Kim and D. J. Wu. Access control encryption for general policies from standard assumptions. In *Advances in Cryptology—Asiacrypt*, 2017.

[40] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for Standard OS Abstractions. In *ACM Symposium on Operating Systems Principles*, 2007.

[41] A. Lewko, T. Okamoto, A. Sahai, K. Takashima, and B. Waters. Fully secure functional encryption: Attribute-based encryption and (hierarchical) inner product encryption. In *Advances in Cryptology—Eurocrypt*, 2010.

[42] A. Lewko and B. Waters. Decentralizing attribute-based encryption. In *Advances in Cryptology—Eurocrypt*, 2011.

[43] J. Liu, O. Arden, M. D. George, and A. C. Myers. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security*, 25(4–5), 2017.

[44] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *ACM Symposium on Operating System Principles*, 2009.

[45] Y. Liu, Y. Guan, X. Li, R. Wang, and J. Zhang. Formal analysis and verification of DDS in ROS2. In *International Conference on Formal Methods and Models for System Design*, 2018.

[46] V. Mayoral-Vilches, R. White, G. Caiazza, and M. Arguedas. SROS2: Usable Cyber Security Tools for ROS 2, 2022.

[47] J. McClean, C. Stull, C. Farrar, and D. Mascareñas. A Preliminary Cyber-Physical Security Assessment of the Robot Operating System. *Unmanned Systems Technology XV*, 8741, 2013. International Society for Optics and Photonics.

[48] `mcl`—A portable and fast pairing-based cryptography library. https://github.com/herumi/mcl.

[49] A. C. Myers. JFlow: practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages*, 1999.

[50] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *ACM Symposium on Operating System Principles*, 1997.

[51] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *IEEE Symposium on Security and Privacy*, 1998.

[52] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4), 2000.

[53] A. Nadkarni, B. Andow, W. Enck, and S. Jha. Practical DIFC enforcement on Android. In *USENIX Security Symposium*, 2016.

[54] NVidia Isaac ROS. https://developer.nvidia.com/isaac-ros.

[55] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: An Open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3.2, 2009.

[56] F. J. Rodríguez-Lera, V. Matellán-Olivera, J. Balsa-Comerón, Á. M. Guerrero-Higueras, and C. Fernández-Llamas. Message Encryption in Robot Operating System: Collateral Effects of Hardening Mobile Robots. *Frontiers in ICT*, 5, 2018.

[57] ROS1 Technical Overview. http://wiki.ros.org/ROS/TechnicalOverview.

[58] ROS2–ROS2 documentation, the latest version of the robot operating system. https://index.ros.org/doc/ros2/.

[59] Y. Rouselakis and B. Waters. Efficient statically-secure large-universe multi-authority attribute-based encryption. In *Financial Cryptography and Data Security*, 2015.

[60] The Real-time Publish-Subscribe Protocol DDS Interoperability Wire Protocol. https://www.omg.org/spec/DDSI-RTPS/2.3/PDF.

[61] A. Sahai and B. Waters. Fuzzy identity-based encryption. In *Advances in Cryptology—Eurocrypt*, 2005.

[62] Setting up security – ROS2 documentation. https://docs.ros.org/en/rolling/Tutorials/Advanced/Security/Introducing-ros2-security.html.

[63] D. Stefan, D. Mazieres, J. C. Mitchell, and A. Russo. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming*, 27, 2017.

[64] Y. Sun, G. Petracca, X. Ge, and T. Jaeger. Pileus: Protecting User Resources from Vulnerable Cloud Services. In *Annual Computer Security Applications Conference*, 2016.

[65] G. Tan, R. Zhang, H. Ma, and Y. Tao. Access Control Encryption Based on LWE. In *ACM International Workshop on ASIA Public Key Cryptography*, 2017.

[66] R. Toris, C. Shue, and S. Chernova. Message authentication codes for secure remote non-native client connections to ROS enabled robots. In *IEEE International Conference on Technologies for Practical Robot Applications*, 2014.

[67] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security Symposium*, 2019.

[68] F. Vercauteren. Optimal pairings. *IEEE Transactions on Information Theory*, 56, 2010.

[69] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles*, 1993.

[70] B. Waters, H. Wee, and D. Wu. Multi-authority ABE from lattices without random oracles. In *Theory of Cryptography Conference*, 2022.

[71] R. White and M. Arguedas. ROS2 Security Enclaves, May 2020. https://design.ros2.org/articles/ros2_security_enclaves.html.

[72] R. White, G. Caiazza, H. Christensen, and A. Cortesi. Procedurally provisioned access control for robotic systems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2018.

[73] R. White, G. Caiazza, C. Jiang, X. Ou, Z. Yang, A. Cortesi, and H. Christensen. Network Reconnaissance and Vulnerability Excavation of Secure DDS Systems. In *IEEE European Symposium on Security and Privacy Workshops*, 2019.

[74] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Maziéres. Making information flow explicit in HiStar. In *Symposium on Operating Systems Design and Implementation*, 2006.

[75] N. Zeldovich, S. Boyd-Wickizer, and D. Maziéres. Securing distributed systems with information flow control. In *Symposium on Networked System Design and Implementation*, 2008.

Appendix—ABE Implementation

In this section, we provide details on the implementation of the Lewko-Waters ABE scheme adapted to our setting. Our description follows the API of the ABE scheme as introduced in Section III.

$\boxed{\mathsf{GlobalSetup}(\lambda)\rightarrow\mathsf{GP}}$

The global setup function takes a security parameter and outputs the global parameters: a prime p and a bilinear pairing $e : G_1 \times G_2 \rightarrow G_T$ where G_1, G_2, G_T are all cyclic groups of prime order p. We use $\bullet$ to denote the multiplicative operator on these groups. Recall that every non-identity element in a cyclic group of prime-order is a generator. We randomly choose generators $g_1 \in G_1$ and $g_2 \in G_2$. We also choose a hash function $\mathcal{H}$: $\mathsf{GID} \rightarrow G_1$ which maps the globally-unique identifiers GID used by the Lewko-Waters scheme to elements in G_1. Because the value $\mathcal{H}(\mathsf{GID})$ is an element of the group G_1, note that it can be expressed in terms of the generator of the group as g_1^{Ω} for some integer Ω.

The groups G_1, G_2, and G_T must be chosen so that a bilinear pairing exists. Prior work has defined such groups and pairings over suitably-chosen elliptic curves [8, 68]. In our implementation, the groups G_1, G_2, and G_T, as well as the bilinear pairing are defined using an elliptic curve—the `BN-254` curve with optimal Ate pairings.

In more detail, the `BN254` curve is defined as the solution set over $F_\rho \times F_\rho$ of the curve $\mathcal{E}$: $y^2{=}x^3{+}b$ for specific values of b and ρ. Note that F_ρ is the base finite field with ρ elements over which the curve is defined. The three groups G_1, G_2 and G_T are defined as follows:

► The group G_1 is the cyclic subgroup of the elliptic curve group itself (of the curve $\mathcal{E}$).

► G_2 is derived from the curve as a cyclic subgroup present in a specific twisting isomorphism on $\mathcal{E}$.

► G_T is the (multiplicative) cyclic subgroup of a certain 12^{th} degree field extension of F_ρ.

For detailed constructions of the groups G_1, G_2, G_T and the bilinear pairing, we refer the reader to the papers by Barreto and Nehrig [8] and Vercauteren [68]. We only provide some details above for completeness. Our implementation simply uses the details of the `BN254` curve directly as provided in `mcl`.

The specific values of b and ρ used for `BN254` yield groups G_1, G_2 and G_T whose size is 0x2523648240000001ba344d80-00000007ff9f800000000010a10000000000000d. For the hash function $\mathcal{H}$, we used the one implemented in the `mcl` library [48]. This hash function follows the specification defined in the Internet Draft [31] and is specifically designed to have its range map into the group elements in G_1.

$\boxed{\mathsf{AuthSetup}(\mathsf{GP},\ \alpha)\rightarrow\{\mathsf{PrivK}_\alpha, \mathsf{PubK}_\alpha\}}$

The $\mathsf{AuthSetup}$ function is invoked once by each authority for each attribute α that it owns. The authority chooses private parameters β_α and $y_\alpha \in \mathbb{Z}_p$. It outputs $\mathsf{PubK}_\alpha = (e(g_1, g_2)^{\beta_\alpha}, g_2^{y_\alpha})$ as the public key corresponding to the attribute α. It privately stores $\mathsf{PrivK}_\alpha = (\beta_\alpha, y_\alpha)$ as the corresponding private key. In the following discussion, we will use the following notation to refer to the two components of the public key: $\mathsf{PubK}_{(\alpha,0)} = e(g_1, g_2)^{\beta_\alpha}$ and $\mathsf{PubK}_{(\alpha,1)} = g_2^{y_\alpha}$.

$\boxed{\mathsf{Keygen}(\mathsf{GID},\ \mathsf{GP},\ \alpha,\ \mathsf{PrivK}_\alpha)\rightarrow\mathsf{K}_{(\alpha,\mathsf{GID})}}$

This attribute-specific and user-specific key $\mathsf{K}_{(\alpha,\mathsf{GID})}$ is given by $g_1^{\beta_\alpha} \bullet \mathcal{H}(\mathsf{GID})^{y_\alpha}$.

$\boxed{\mathsf{Encrypt}(M,\ \mathsf{GP},\ \{\mathsf{PubK}_\alpha\},\ \mathcal{A})\rightarrow C}$

The encryption algorithm takes in a message $M \in G_T$, the global parameters, an access structure $\mathcal{A}$ over the set of attributes and the set of public keys $\{\mathsf{PubK}_\alpha\}$ corresponding to the attributes appearing in $\mathcal{A}$. For the case where $\mathcal{A}$ is of the form $\mathcal{A} = \bigwedge_{i=1}^{n} \alpha_i$, *i.e.,*a Boolean conjunction over n attributes, let S be the set of corresponding public keys $S = \{\mathsf{PubK}_{\alpha_i} \mid i \in \{1, 2, \cdots n\}\}$ with $|S| = n$.

Let $\mathcal{K}^{(n)}$ be the following $n \times n$ matrix. We describe the significance of this matrix in the Lewko-Waters construction later in this section.

$$\mathcal{K}^{(n)} = \begin{pmatrix} 1 & -1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & -1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & -1 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & & & & & \vdots & \\ 0 & 0 & 0 & 0 & \cdots & 1 & -1 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 \end{pmatrix}$$

The encryption algorithm chooses:

► A random $\sigma \in \mathbb{Z}_p$;

► A random vector $v \in \mathbb{Z}_p^n$ with σ as its first entry. It uses this vector v to compute $\lambda_i = \langle \mathcal{K}_i^{(n)}, v \rangle$ (*i.e.,* the inner product), where $\mathcal{K}_i^{(n)}$ is the i^{th} row of $\mathcal{K}^{(n)}$;

► A random vector $w \in \mathbb{Z}_p^n$ with 0 as its first entry. It uses w to compute $w_i = \langle \mathcal{K}_i^{(n)}, w \rangle$, where $\mathcal{K}_i^{(n)}$ is the i^{th} row, as above.

► A random integer $r_i \in \mathbb{Z}_p$ for each row $\mathcal{K}_i^{(n)}$.

Observe that the matrix $\mathcal{K}^{(n)}$ is constructed in such a way that $\sum_{i=1}^{n} \mathcal{K}_i^{(n)} = (1, 0, 0, \cdots, 0)_{1\times n}$. As a result, the following two properties hold: $\sum_{i=1}^{n} \lambda_i = \sigma$ and $\sum_{i=1}^{n} w_i = 0$. Now define the following terms:

► $C_0 = M \bullet e(g_1, g_2)^{\sigma}$;

► $C_{(1,i)} = e(g_1, g_2)^{\lambda_i} \bullet \mathsf{PubK}_{(\alpha_i,0)}^{r_i} = e(g_1, g_2)^{\lambda_i} \bullet e(g_1,\ g_2)^{\beta_{\alpha_i} \cdot r_i}$, computed for each $i \in \{1, 2, \cdots n\}$ (*i.e.,* for each row of the matrix $\mathcal{K}^{(n)}$);

► $C_{(2,i)} = g_2^{r_i}$ for each $i \in \{1, 2, \cdots n\}$; and

► $C_{(3,i)} = g_2^{w_i} \cdot \mathsf{PubK}_{(\alpha_i,1)}^{r_i} = g_2^{w_i} \bullet g_2^{y_{\alpha_i} \cdot r_i} = g_2^{w_i + y_{\alpha_i} \cdot r_i}$ for each $i \in \{1, 2, \cdots n\}$.

The ciphertext computed by the Encrypt function is a tuple with the $3n + 1$ terms $C = [C_0,\ \forall_{i=1}^{n} C_{(1,i)},\ \forall_{i=1}^{n} C_{(2,i)}\ \forall_{i=1}^{n} C_{(3,i)}]$ described above.

$\boxed{\mathsf{Decrypt}(C,\ \mathsf{GP},\ \{\mathsf{K}_{(\alpha,GID)}\}) \rightarrow M}$

We assume the entity that invokes this function has all the keys required. The decryptor first computes $\mathcal{H}(\mathsf{GID})$ and then for each row i in $\mathcal{K}^{(n)}$, computes the following values for $i \in \{1, 2, \cdots n\}$:

$$\frac{C_{(1,i)} \bullet e(\mathcal{H}(\mathsf{GID}), C_{(3,i)})}{e(\mathsf{K}_{(\alpha_i,\mathsf{GID})}, C_{(2,i)})}$$

We now claim:

$$\frac{C_{(1,i)} \bullet e(\mathcal{H}(\mathsf{GID}), C_{(3,i)})}{e(\mathsf{K}_{(\alpha_i,\mathsf{GID})}, C_{(2,i)})} = e(g_1, g_2)^{\lambda_i} \bullet e(\mathcal{H}(\mathsf{GID}), g_2)^{w_i}$$

Proof: Recall that $\mathcal{H}(\mathsf{GID}) = g_1^{\Omega}$ for some integer Ω, and $C_{(3,i)} = g_2^{w_i + y_{\alpha_i} . r_i}$. Thus,

$$e(\mathcal{H}(\mathsf{GID}), C_{(3,i)}) = e(g_1^{\Omega}, g_2^{w_i + y_{\alpha_i} \cdot r_i}) = e(g_1, g_2)^{\Omega \cdot w_i + \Omega \cdot y_{\alpha_i} \cdot r_i}$$

where the last equality follows due to the bilinearity property. Similarly,

$$e(\mathsf{K}_{(\alpha_i,\mathsf{GID})}, C_{(2,i)}) = e(g_1^{\beta_{\alpha_i}} \bullet (g_1^{\Omega})^{y_{\alpha_i}}, g_2^{r_i}) = e(g_1^{\beta_{\alpha_i} + \Omega \cdot y_{\alpha_i}}, g_2^{r_i})$$

Applying bilinearity, we get:

$$e(\mathsf{K}_{(\alpha_i,\mathsf{GID})}, C_{(2,i)}) = e(g_1, g_2)^{\beta_{\alpha_i} \cdot r_i + \Omega \cdot y_{\alpha_i} . r_i}$$

Therefore we have:

$$\frac{C_{(1,i)} \bullet e(\mathcal{H}(\mathsf{GID}), C_{(3,i)})}{e(\mathsf{K}_{(\alpha_i,\mathsf{GID})}, C_{(2,i)})} = \frac{C_{(1,i)} \bullet e(g_1, g_2)^{\Omega \cdot w_i + \Omega \cdot y_{\alpha_i} \cdot r_i}}{e(g_1, g_2)^{\beta_{\alpha_i} \cdot r_i + \Omega \cdot y_{\alpha_i} \cdot r_i}}$$

$$= C_{(1,i)} \bullet e(g_1, g_2)^{\Omega \cdot w_i - \beta_{\alpha_i} \cdot r_i}$$

$$= e(g_1,\ g_2)^{\lambda_i} \bullet e(g_1, g_2)^{\beta_{\alpha_i} \cdot r_i} \bullet e(g_1, g_2)^{\Omega \cdot w_i - \beta_{\alpha_i} \cdot r_i}$$

$$= e(g_1,\ g_2)^{\lambda_i} \bullet e(g_1, g_2)^{\Omega \cdot w_i} = e(g_1,\ g_2)^{\lambda_i} \bullet e(g_1^{\Omega}, g_2)^{w_i}$$

$$= e(g_1,\ g_2)^{\lambda_i} \bullet e(\mathcal{H}(\mathsf{GID}), g_2)^{w_i} \text{ since } \mathcal{H}(\mathsf{GID}) = g_1^{\Omega} \quad \blacksquare$$

With this identity in hand, the decryptor computes the product of all the terms

$$\Pi_{i=1}^{n} \frac{C_{(1,i)} \bullet e(\mathcal{H}(\mathsf{GID}), C_{(3,i)})}{e(\mathsf{K}_{(\alpha_i,\mathsf{GID})}, C_{(2,i)})}$$

By the claim above this product is:

$$\Pi_{i=1}^{n} e(g_1, g_2)^{\lambda_i} \bullet e(\mathcal{H}(\mathsf{GID}), g_2)^{w_i}$$

$$= e(g_1, g_2)^{\sum_{i=1}^{n} \lambda_i} \bullet e(\mathcal{H}(\mathsf{GID}), g_2)^{\sum_{i=1}^{n} w_i}$$

Using the observation that $\sum_{i=1}^{n} \lambda_i = \sigma$ and $\sum_{i=1}^{n} w_i = 0$, the above term becomes $e(g_1, g_2)^{\sigma}$. The plaintext is then obtained:

$$M = \frac{C_0}{e(g_1, g_2)^{\sigma}}$$

As presented thus far, the construction of the ABE primitives applies only for the special case where the access structure is a conjunction of all the attributes. However, the original Lewko-Waters construction [42] allows for more general access structures than just conjunctions, namely, those that can be represented by Linear Secret Sharing Scheme (LSSS) access matrices. A LSSS access matrix is a $l \times m$ matrix with entries in $\mathbb{Z}_p$ along with a mapping Ψ which maps the rows of the matrix to the attributes. An attribute set S is authorized under the access matrix if the row vector $(1, 0, \cdots, 0)_{1\times m}$ lies in the span of the rows $\Psi^{-1}(S)$ *i.e.,* the set of all rows to which the attributes in S are mapped. This a generalization of the condition under which an attribute set satisfies a Boolean circuit. The intuition behind this construction is that when a user's attribute set is authorized, they can create an appropriate product so as to obtain the term $e(g_1, g_2)^{\sigma}$. Note that the vectors v and w are carefully chosen so that their first coordinate is σ and 0, respectively.

Therefore to create a LSSS access matrix for a Boolean gate comprising a simple conjunction of up to n attributes (as is required in Picaros), we need to create a matrix for which the row $(1, 0, \cdots, 0)_{1\times m}$ is in the span of all the n rows of the matrix and *not* in the span of any proper subset of the rows. The matrix $\mathcal{K}^{(n)}$ is one such matrix that satisfies this requirement. Note that $\mathcal{K}^{(n)}$ is not the only matrix satisfying this case. But $\mathcal{K}^{(n)}$ provides some useful computational benefits:

► In $\mathcal{K}^{(n)}$, the appropriate linear combination to get the vector $(1, 0, \cdots, 0)_{1\times m}$ is just the sum of all the rows, hence the corresponding product requires no modular exponentiations from the decryptor and is computationally faster.

► When the attribute set contains a single attribute (*i.e.,* the message is encrypted with the encryption key belonging to a single attribute), the matrix contains a single element, while λ_1 and w_1 are simply σ and 0, respectively.

A general algorithm to create such LSSS matrices for arbitrary Boolean formulae is provided in the appendix of the Lewko-Waters paper [42].