Privaros: A Framework for Privacy-Compliant Delivery Drones

Rakesh Rajan Beck
Indian Institute of Science
rakeshbeck@iisc.ac.in

Abhishek Vijeev*
Indian Institute of Science
abhishekvijeev@iisc.ac.in

Vinod Ganapathy
Indian Institute of Science
vg@iisc.ac.in

ABSTRACT

We present Privaros, a framework to enforce privacy policies on drones. Privaros is designed for commercial delivery drones, such as the ones that will likely be used by Amazon Prime Air. Such drones visit various host airspaces, each of which may have different privacy requirements. Privaros uses mandatory access control to enforce the policies of these hosts on guest delivery drones. Privaros is tailored for ROS, a middleware popular in many drone platforms. This paper presents the design and implementation of Privaros's policy-enforcement mechanisms, describes how policies are specified, and shows that policy specification can be integrated with India's Digital Sky portal. Our evaluation shows that a drone running Privaros can robustly enforce various privacy policies specified by hosts, and that its core mechanisms only marginally increase communication latency and power consumption.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**.

KEYWORDS

ROS; drones; privacy; mandatory access control; trusted computing

ACM Reference Format:
Rakesh Rajan Beck, Abhishek Vijeev, and Vinod Ganapathy. 2020. Privaros: A Framework for Privacy-Compliant Delivery Drones. In *2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3372297.3417858

1 INTRODUCTION

Over the past few years, there has been a rapid increase in the availability and ownership of end-user drones. Drones are now available for a few hundred dollars and widely used by hobbyists. Commercial operators such as Amazon are also planning to use fleets of drones for delivery. The US Federal Aviation Administration forecast report (2019-2039) [1] predicts over 1.39 million hobbyist and 853,000 commercial drones by 2023. Drones are also being put to effective use in the Covid19 pandemic. Law-enforcement agencies in various countries are using drones to make public-service announcements, and patrol locked-down areas for unauthorized social gatherings.

Despite the novel applications enabled by drones, the lack of tight regulations surrounding their use has led to a plethora of security and privacy problems. Incidents involving drones range from potential drone/aircraft collisions and near-misses [95], drone-sightings causing airport closures [22], to smuggling [21] and assassination attempts [45]. While rogue drones cause such security and safety-related problems, benign drones, *e.g.,* those that may be used for package delivery, also raise serious privacy concerns. Drones are equipped with a variety of sensors (cameras, GPS, Lidar, *etc.*) for navigation. The sensors on board the drone can be used to capture pictures or video, map a sensitive location or a building. Prior studies have shown that people are indeed wary of their privacy being compromised by drones [16, 89, 94]. Addressing the entire gamut of security and privacy problems posed by drones requires new regulations (*e.g.,* to ensure that drones have an identity registered with the aviation authority), technology, and law enforcement (*e.g.,* to detect [11, 14, 15, 33, 61, 73, 81] and capture rogue drones).

We present **Privaros**, a framework that allows *host airspaces* (*e.g.,* a corporate or university campus, a city neighbourhood, or an apartment complex) to ensure that *guest drones* entering them are compliant with privacy-policies determined by the hosts. For example, a host can specify a policy that requires any guest drone that enters its airspace to refrain from wirelessly transmitting or locally storing (*e.g.,* in an on-board SD card) any images or video that it captures when within the host's airspace. Privaros enhances the drone software stack with mechanisms that allow guest drones to enforce host-specified privacy policies and prove to the host that they are in compliance (via hardware-based attestations).

We have designed Privaros specifically with a focus on delivery drones. These drones are managed by fleet operators that have reputations and delivery contracts to protect and, by corollary, have no incentives to operate rogue drones. Thus, we can assume that such drones have an identity (*e.g.,* a public key) that is registered with the aviation authority, are equipped with the Privaros-enhanced software stack, and have associated trusted hardware that makes remote attestation possible. Making these assumptions allows us to focus on the key challenges in building the policy-compliance mechanisms for drones without rightaway having to consider other critical issues, *e.g.,* on how to issue identity to drones and on how to deal with rogue drones. While central to an end-to-end treatment of privacy with drones, these issues involve developing regulations and evolving new law-enforcement methods that are outside the scope of this paper. Our work focuses on delivery drones because their usage model implies that the above assumptions hold. Our work also gels well with the policies being developed by various countries [6, 9, 40, 54, 62], notably India's Digital Sky [37–39], which provides guidelines for drone operators in India.

Privaros models the problem of enforcing host-specified privacy policies as one of regulating how applications on the drone consume or communicate data received from sensors on the drone. Privaros

*Rakesh Rajan Beck and Abhishek Vijeev contributed equally to this research.

CCS'20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417858

enforces these restrictions using mandatory access control. For the example policy discussed earlier, Privaros can ensure that the video feed from the camera is available to image-processing/vision applications and to the navigation software, but cannot be sent to local storage or to the wireless network interface.

Privaros is built on top of the Robot Operating System [67, 71, 72] (ROS version 2[1]), a popular middleware used by a number of drones (*e.g.,* various models sold by DJI, 3DR, Parrot, Gaitech, Erle, BitCraze, Skybotix) and other robotics systems. A key reason for our choice to base Privaros atop ROS was its rich support for applications written in a variety of languages, including Python and C++. ROS provides the abstractions to transparently execute a variety of applications on any drone hardware platform that runs ROS, and also interacts with the navigation control software. A vibrant application ecosystem has evolved around ROS and the market for platforms that use ROS is expected to grow to $402.7 million by 2026 [68]. Privaros can therefore directly benefit drone operators that tap into the ROS application ecosystem.

ROS is built as a publish/subscribe system, in which ROS applications publish or subscribe to certain *topics*. ROS simply acts as a matchmaker that pairs publishers and subscribers, following which the pair of applications communicate directly with each other over network sockets. As such, ROS does not incorporate any security mechanisms to regulate application communication. Thus, a malicious ROS application can easily snoop on or corrupt the communication between a pair of benign applications. Recognizing the need to prevent such attacks, the ROS community has developed Secure ROS (SROS) [90, 91], a set of extensions that attempt to prevent such attacks.

In this paper, we show that the mechanisms of SROS alone do not suffice to robustly enforce security policies. In particular, while ROS applications typically communicate via the publish/subscribe mechanism, they can also communicate directly via other operating system (OS) abstractions, such as raw sockets, shared memory, pipes, and the file system. For example, a pair of applications can bypass the ROS-based publish/subscribe matchmaking mechanism, and directly establish socket connections for communication. While ROS has visibility into the publish/subscribe system and can reason about applications that initiate communication using this system, it cannot reason about low-level communication via OS abstractions.

Privaros enhances the ROS software stack by adding the ability to enforce mandatory access control policies between ROS applications. It tightly integrates policy enforcement at the ROS layer with OS kernel-level modifications to enforce mandatory access-control policies. At the OS level, the mechanisms of Privaros allow it to robustly enforce restrictions on applications that communicate directly via OS abstractions or bypass ROS. At the ROS level, Privaros incorporates modifications that allow the OS mechanisms to be cognizant of ROS abstractions (*e.g.,* topic names) used by applications, and suitably redirect communication via trusted applications, where required. Section 3 elaborates on the mechanisms in Privaros.

We have tailored Privaros's policy interface for delivery drone operations in India. To show that drones using Privaros can readily be adopted once regulations are in place, we integrate Privaros's policy specification interface with the front-end offered by India's Digital Sky portal [37]. This interface allows drone operators to specify the regions to which they intend to fly and obtain the permission to fly from India's Directorate General of Civil Aviation (DGCA). While the current intent of Digital Sky's interface is to prohibit drones from flying over so-called "red-zones" (*e.g.,* military establishments or other sensitive areas), we show that the same interface can be used to upload the privacy policies of all the host airspaces that the drone will visit during its delivery run (Section 4). As a result, we hope that Privaros can be readily adopted without invasive changes to a regulatory platform that is already in place.

For our experiments, we ran the Privaros-enhanced software stack on an NVidia Jetson TX2 board [80]. We chose this platform because its firmware can easily be reflashed with Privaros (unlike off-the-shelf drones, which are often closed platforms), and also because it offers a programmable trusted-execution environment (TEE) based on the ARM TrustZone [5]. Commercially-available drones do not yet have the kind of hardware support to enable remote attestations by hosts. However, we note that such trusted hardware support has already been proposed as part of the regulations in Digital Sky [39]. Indian drone vendors will thus have to provide trusted hardware in the near future to sell and operate drones in India. Our evaluation (Section 5) shows that Privaros robustly enforces privacy policies. Furthermore, Privaros's core mechanisms only introduce low runtime overheads in terms of communication latency and power consumption on the drone.

To summarize our contributions:

- We motivate the problem of enforcing host-specified privacy policies on guest delivery drones and discuss the shortcomings of existing methods to enforce such policies;
- We present Privaros, a set of new mechanisms added to ROS and the underlying OS to enforce such policies;
- We show how the policy specification for Privaros can be integrated with the Digital Sky interface; and
- We present a robustness and security evaluation of Privaros on an NVidia Jetson TX2 board, and a performance evaluation showing that its overheads are low enough for practical use.

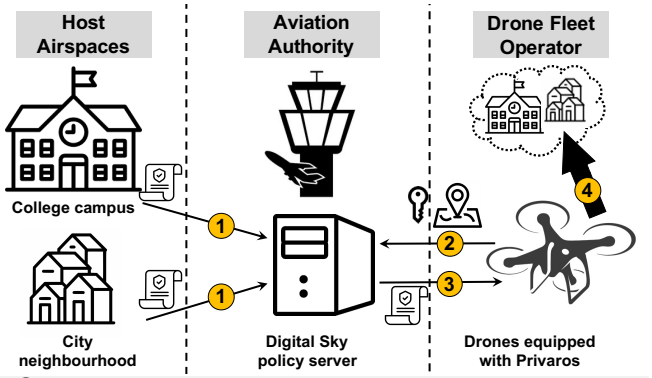
2 BACKGROUND AND THREAT MODEL

We now present examples of the kinds of policies that we consider in this paper, background on the regulations that have already been proposed by Indian aviation authorities, and our threat model.

Example policies. A host airspace may wish to impose a variety of policies on guest drones:

① ProcessLocally. Autonomously-navigated drones capture images or video of their surroundings. These images/video are processed on-board by a computer vision application to detect obstacles that the drone must fly around. A host may wish to ensure that the images/video captured by such a drone are only used by the computer vision application, which in turn communicates this information only to the navigation board. In particular, the images/video must not be transmitted outside the drone via its network interfaces. They should also not be stored in the drone's filesystem for retrieval by the drone operator at a later point in time.

[1]Unless otherwise noted, uses of the term "ROS" in this paper reference ROS version 2 [72], which deviates significantly in design from ROS version 1 [71].



① Host airspaces, such as college/school campuses, office complexes/IT parks, and city neighbourhoods, identify their privacy policies and upload it to the aviation authority (Digital Sky in India); ② A delivery drone operator starting up a drone sends information identifying the drone (*e.g.,* its registered public key), and declares the flight path of the drone's delivery run; ③ The aviation authority server vets the flight plan (*e.g.,* for NPNT compliance) and then sends the drone the privacy policies of the host airspaces along its flight path; ④ The drone then begins its delivery run. Any host airspace along the drone's flight path can challenge it to provide TEE-backed attestations, to verify the integrity of the drone's Privaros-enabled software stack.

Figure 1: Deployment scenario for Privaros-enabled drones.

② BlurExportedImages. A large majority of drones available today are controlled by a ground-based operator. These drones transmit a video feed from the drone to the operator (called the *first person view*) who navigates the drone manually, often with visual line of sight. Alternatively, one could imagine an autonomously-navigated drone that transmits its video feed to a cloud-based server for obstacle detection, and obtains its navigation decisions from this server. For such drones, it is impractical for hosts to impose the ProcessLocally policy. Hosts may instead wish to ensure that the video feed exported from the drone is scrubbed to remove sensitive information. For example, the host may require the video feed to be processed by an on-board application (vetted by the host) that blurs peoples' faces and car registration plates that appear in the video feed (*e.g.,* as is done in images published in Google Street View).

③ UseDroneLanes. The host may require guest drones to fly only within designated *drone lanes* to ensure safety and privacy. In a campus or university setting, campus security may identify drone lanes that are away from sensitive installations within the campus. Localities in a city may likewise identify drone lanes that border public spaces at a comfortable distance away from private homes.

These policies have been discussed in prior work [48, 83], but are by no means a comprehensive listing of policies that Privaros can enforce. Hosts may choose different combinations of these or other policies to enforce in their airspaces. The policies may also have to be tailored to the abilities of the guest drone, *e.g.,* an autonomous drone can operate under ProcessLocally, but a semi-automated or manual drone may require BlurExportedImages.

These example policies cover the main concerns that have been raised in drone privacy laws proposed in several countries, *e.g.,* the United States [9, 54], United Kingdom [6, 40], Australia [62], and India [38, 39], where privacy of individuals and the integrity of the drone's flight path are the primary considerations. None of these proposals suggest a way to *enforce* these policies beyond seeking an individual's permission before recording their picture or video.

Digital Sky. The Digital Sky platform is a set of regulations [38, 39] and supporting computing infrastructure [37] that India's aviation authority (DGCA) is using to formulate its drone policy. A first set of regulations [38] was adopted on December 1, 2018 as part of the Civil Aviation Regulations, and the policy continues to evolve. Aimed primarily at drone operators, the Digital Sky portal [37] provides interfaces for authorized users (*e.g.,* licensed commercial drone operators) to register the identity of their drones with DGCA, and obtain permission to fly before each delivery run.

Digital Sky's policy is focused on ensuring safety and security. For example, Digital Sky allows operators to specify the geographic region over which they intend to fly using a visual map-based interface. This information is sent to a back-end server, which checks whether the region intersects any red-zones (*e.g.,* sensitive military installations), which are no-fly zones. The proposed flight path may also intersect airspaces that impose altitude restrictions. Drones are not allowed to fly above a certain altitude as they approach an airport, for instance, and the permitted altitude reduces as the drone approaches the airport (thereby imposing a conical exclusion zone centred at the airport).

One of the centrepieces of Digital Sky's proposal to enforce these restrictions is *No-permission No-takeoff (NPNT).* With NPNT, the DGCA server would review the drone's flight path and issue a permission artifact, digitally-signed by the aviation authority. This permission artifact is sent to the drone, and the drone is permitted to fly only after validating the digitally-signed permission artifact.

Finally, Digital Sky certifies drones at various levels [39, Page 39], based on the hardware capabilities of the drone. A level 0-certified drone stores cryptographic artifacts pertaining to its identity (*i.e.,* its public/private key pair) completely within software. A level 1-certified drone must have a hardware-based TEE to store the drone's private keys, perform attestations, and perform NPNT validation and enforcement. Level 1-certified drones are robust to attacks on the drone's software stack because they offer a hardware-only TCB.

Privaros aims to build upon the basic protections offered by Digital Sky by providing fine-grained policy enforcement within the drone. Privaros allows enforcement of policies such as ProcessLocally, BlurExportedImages and UseDroneLanes that are beyond the current scope of Digital Sky. Figure 1 depicts how we envision Privaros-enabled drones to be deployed, and how policies will be communicated to the drones. Section 4 presents our deployment vision in more detail.

Threat model. Privaros is tailored for delivery drones. E-commerce companies considering drone-based delivery (*e.g.,* Amazon, Flipkart) will likely use decentralized models akin to those used in ground-based delivery, wherein procurement and operation of delivery vehicles is outsourced to delivery-service providers (DSPs) [20, 29, 49]. While it is reasonable to assume that e-commerce companies are trusted and have no overtly-malicious intentions (because they have reputations to protect), host airspaces may not trust DSPs.

In particular, the e-commerce company may prescribe a Privaros-enhanced software stack for use on a DSP's delivery drones. However, the host airspace cannot trust that the DSP's guest drone is indeed executing that software stack. For example, the drone may have been compromised by a rogue DSP employee who covertly reflashes the drone's software or installs malware on it.

We therefore require guest drones to attest their software stack to host airspaces. Our trusted-computing base (TCB) consists of the guest drone's hardware TEE and its OS, enhanced with the mechanisms of Privaros. The goal of attesting the drone is to ensure that the TCB in the guest drone is untampered. Applications running on top of Privaros are not trusted and could be malicious in intent. Privaros also introduces modifications to ROS (Section 3.3.3). However, ROS consists of library modules that are linked against applications. Since we do not trust applications, the modifications that Privaros introduces in ROS are not part of the TCB.

To enable attestations, we assume that each drone is equipped with a hardware TEE (*i.e.,* a level 1-certified drone in Digital Sky terminology) that stores the drone's private key. The drone's public key serves as its identity to hosts and the aviation authority. The hardware TEE enables features like secure boot and attestations in response to challenges from hosts or the aviation authority. We have implemented Privaros on a hardware platform that has a TEE based on ARM TrustZone [5]. However, Privaros only requires a TEE to attest the software stack (in the standard way [75]) and its design does not currently leverage many other features of the TrustZone (*e.g.,* peripheral partitioning across worlds). It can therefore be adapted to any hardware TEE design that drone vendors evolve in response to Digital Sky regulations.

A single drone typically has multiple compute platforms. For example, a *flight control board* (such as Pixhawk [64]) runs the autopilot software (such as Ardupilot [4] or PX4 [66], running on top of a real-time operating system such as ChibiOS [18]) and interacts with a *companion board* (that typically uses an ARM-based processor) that runs applications. We assume that attestations provided by the drone cover the software running on *all* these compute platforms. This could be implemented with a single *master* board (*i.e.,* the registered flight module whose identity is provided to the aviation authority) obtaining local attestations from all *slave* boards, and providing a consolidated attestation to an external entity such as a host. Digital Sky requires all master-slave communications to be encrypted with 128-bit symmetric keys, at minimum [39, Page 39]. For now, we only obtain attestations from the companion board that executes Privaros. This is primarily because flight control boards are not currently equipped with hardware TEEs, although they are likely to evolve to be equipped as such.

By corollary, our threat model excludes physical attacks that attempt to bypass the mechanisms of Privaros. For example, a rogue employee could attempt to bypass Privaros by clipping on a remote-controlled camera with in-built networking that is not connected as a sensor to the compute platform running Privaros. To an extent, some of these attacks can be mitigated with regulatory compliance checks. For example, regulations may require the fleet operator to have procedures that perform an automated physical check of the drone before it flies out of the warehouse on its delivery run to ensure that there are no unauthorized peripherals on the drone.

```
Message format for topic CamOutput
uint32   height    # image height
uint32   width     # image width
string   encoding  # encoding of pixels
uint32   step      # row length
uint8[]  data      # image matrix (step*height)
```

Figure 2: Example declaration of message format in ROS. To send messages, an application uses a topic name (say CamOutput) to refer to the message stream, and publishes messages in this format under that topic name.

Privaros provides the ability to control how applications consume sensor data. However, it is well-known that mandatory access control (*e.g.,* based on subject and object labels) is a not perfect mechanism. It cannot protect against applications attempting to communicate with each other via covert timing or storage channels. We exclude covert channel-based attacks from our threat model.

3 ENFORCEMENT MECHANISM

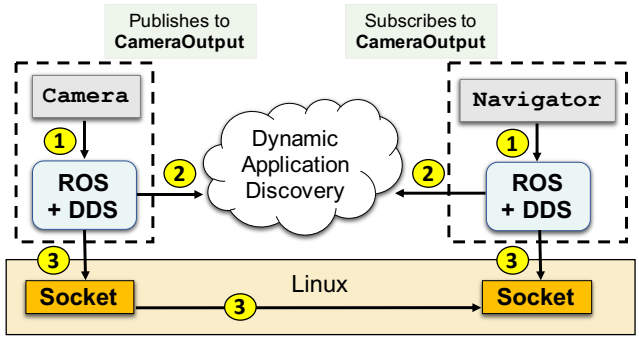
This section presents the details of the policy enforcement mechanism in Privaros. Section 4 will describe how policies are specified and communicated to the drone.

3.1 ROS

Privaros enhances ROS and the underlying OS, and mediates the actions of all the applications running on the drone. As discussed in our threat model, drone applications are typically executed on a companion board. This is standard in all ROS-based platforms. The flight-control board and the sensors connected to it communicate with applications on the companion board using the MAVLink protocol [51]. Applications receive and process data from drone sensors and can also communicate with each other. For example, the output of the camera can be processed by an image-processing pipeline to detect obstacles. The output from this pipeline may be processed by a navigation application that sends MAVLink control commands to the flight-control board. Privaros aims to control inter-application communication based on the host's privacy policies.

ROS primarily uses a publish/subscribe model to facilitate application communication. The publish/subscribe mechanism in ROS is built using the Data Distribution Service (DDS) [24, 74], an open middleware standard created for real-time and embedded systems. ROS enables asynchronous communication between applications while decoupling spatial and temporal concerns, *i.e.,* applications don't need to know where other applications that they communicate with reside (they can even run on a different drone), and applications can exchange information even if they are not simultaneously running. Applications publish or subscribe to one or more *topics*, identified by a topic name. Topics have associated types that specify how messages published under that topic must be parsed. Figure 2 presents an example of how an application would specify a ROS topic (in this case, an image). The fields shown in the topic declaration are the various data members of messages that are published under that topic.

ROS uses the DDS protocol to match publishers with subscribers based on topic. At the application level, the abstraction presented is one of publishing messages to a bus, which are delivered to all



① Every application on ROS links against the library. The dotted lines show the process boundary. An application registers its topics via the ROS library; ② A decentralized protocol discovers and identifies applications with matching topics; ③ ROS sets up socket communication via the underlying OS for the applications.

Figure 3: Publisher/subscriber communication in ROS.

subscribers of the bus. Figure 3 illustrates how this abstraction is implemented. When ROS starts an application that subscribes to a topic, it checks which applications publish to that topic; DDS implements a decentralized protocol for application discovery. If it identifies a publisher, it sets up a network socket for the publisher and subscriber to directly communicate with each other. ROS is built to support distributed robotics platforms, so a publisher and subscriber application need not necessarily run on the same physical machine. However, if they do, ROS may choose to optimize their communication using shared memory instead of sockets. Applications that exclusively use the ROS API for communication remain oblivious to the means of communication (sockets, shared memory) that ROS uses to establish communication. The ROS library, which applications are linked with, transparently marshals and unmarshals data beneath the application layer, thereby exposing a simple publish/subscribe API at the application layer.

We use a *communication graph* to represent the flow of messages between applications on a ROS system. The nodes of this graph represent ROS applications while edges denote topic names. Note that each application can publish or subscribe to multiple topics. We present examples of communication graphs in Section 3.3.

3.2 SROS and its Shortcomings

In its most basic form, ROS does not offer security. Applications do not authenticate each other and messages between applications are exchanged in the clear. This leads to a number of attacks [25, 26, 42, 52, 69, 84] that compromise message confidentiality (*e.g.,* snooping on messages), data integrity (*e.g.,* false data injection) and sender integrity (*e.g.,* by impersonating an application).

ROS does not impose restrictions on the topics to which an application can publish or subscribe. This leads to situations where an application can publish a synthetic image feed with the same topic name as the real camera (say, CamOutput is the topic name). Applications that subscribe to the CamOutput topic will consume this image feed, possibly with dire consequences. For example, a malicious application can fool an obstacle-detection application that subscribes to CamOutput by publishing an obstacle-free image

```
<permissions>
   <grant name="/camera">
   <subject_name>CN=/camera</subject_name> ...
   <allow_rule>
      <publish>
      <topics>
         <topic>CamOutput</topic>
         <topic>CameraStatus</topic>
      </topics>
      </publish>
      <subscribe>
      <topics>
         <topic>Clock</topic>
      </topics>
      </subscribe>
   </allow_rule>
   <default>DENY</default>
   </grant>
</permissions>
```

Figure 4: Snippet of an application manifest in SROS.

feed, thereby causing the drone to crash into a building. Similarly, a malicious network-facing application can subscribe to CamOutput and transmit the image feed to the attacker's server.

A number of prior papers have investigated these security shortcomings and have also proposed solutions [25, 26, 52, 69]. The first version of ROS also had a centralized ROS master node, which was responsible for matchmaking. As a centralized entity, its failure could lead to denial-of-service attacks [42]; ROS version 2, which we use for Privaros, eliminates the ROS master node, and instead uses the decentralized DDS protocol for communication setup.

To address these concerns, the community has developed the Secure ROS extension (SROS) [90, 92]. SROS requires each node in the communication graph to be associated with an identity backed by a X.509 certificate, signed by a trusted third-party. SROS secures communication between nodes using TLS. It also allows application-writers to specify a manifest that lists the topics to which that application can publish or subscribe (*e.g.,* see Figure 4). The manifest is cryptographically bound to the application's identity and cannot be modified without regenerating associated the X.509 certificate. SROS thus ensures that an application cannot listen to or produce messages on topics that are not already part of its declared manifest.

These mechanisms prevent a number of basic attacks that are otherwise possible on a ROS system. But they are not perfect, and do not suffice to enforce policies end-to-end. We identify two fundamental, design-level shortcomings:

① *Lack of end-to-end reasoning.* SROS restricts the list of topics to which an application can publish or subscribe via its manifest. However, when an application author specifies this list in the manifest, he does not know *a priori* what other applications will execute on the drone platform. This lack of context-specific, end-to-end reasoning about the data produced or consumed by an application restricts our ability to enforce policies in arbitrary settings. For example, the ProcessLocally policy prevents any images published by the camera (under the topic CamOutput) from being transmitted outside the drone. However, BlurExportedImages does allow images to

leave the drone as long as they are scrubbed by another application to blur any privacy-sensitive data in the images. The application author, who specifies the manifest, has no way to reason about all the contexts in which the application will execute. Without such reasoning about the application's end-to-end usage, the application author can at best produce a one-size-fits-all manifest that may poorly fit the situation in which the application is used.

② *Lack of control over lower-level abstractions.* SROS only imposes constraints on communication that goes via the ROS platform. Applications (both malicious and benign ones) can choose to bypass ROS entirely, and communicate directly with each other via network sockets, shared memory, the file system, or inter-process communication. Such communication happens directly via OS abstractions and therefore completely bypasses SROS enforcement.

In addition to these design-level shortcomings in SROS, we also identified some quirks in its implementation that could lead to unexpected attacks. First, SROS only allows application authors to specify restrictions in the manifest using *topic names*. ROS version 2 allows an application to publish messages belonging to *different types* under the *same topic name*. For example, a camera application publishing under the topic CamOutput could publish images under one type (say, CamOutput::ImageType) and its status under another type (say, CamOutput::StatusType). An application can choose to subscribe to messages of one or more of these types under the same topic. However, the type of data consumed by the application will not be evident in the manifest file, which only specifies the topic. For example, an application called `CameraStatus`[2] could periodically poll the camera's status by subscribing to CamOutput and only read the data value published with type CamOutput::StatusType. The fact that this application does not read the image feed from the camera is not evident from the manifest file.

The second quirk is that SROS internally uses the *full path* of the application binary to identify the application at runtime. Using the path rather than the actual executable to determine identity makes the system vulnerable to attacks where the application binary is replaced with a malicious version. SROS will use the same manifest as the original application to determine the list of topics accessible to the malicious application.

Taken together, these quirks enable a malicious drone operator to engineer data leaks in certain situations. For instance, suppose the `CameraStatus` application is allowed to upload the camera's operational status to the network. A well-behaved `CameraStatus` application only reads data of type CamOutput::StatusType, but not its image feed (of type CamOutput::ImageType). A drone running such an application should therefore be acceptable to a host that wishes to enforce the ProcessLocally policy. However, if SROS were used for policy enforcement, a malicious drone operator could violate the ProcessLocally policy by replacing the `CameraStatus` application binary with a malicious version. The malicious application reads data of type CamOutput::ImageType and leaks it over the network. SROS allows this attack because Ⓐ it only uses the topic name in the manifest file to restrict the data channels accessible to the application; and Ⓑ it only uses the path name of the binary and does not bind the executable to its identity.

[2]Throughout the paper, please note the font conventions used for `Applications`, Topics, versus Policies.

We do not view these implementation-level quirks in SROS to be foundational. Indeed, there are easy workarounds: *e.g.,* Ⓐ modify ROS to include the type name with the topic in the manifest file (or match types at runtime); Ⓑ rewrite applications to decouple different types of data into different topics; and Ⓒ bind the application binary to its identity, possibly coupled with hardware TEE-based attestation of the binary to the host. However, the design-level shortcomings of SROS are the primary motivation for Privaros.

3.3 New Mechanisms in Privaros

Privaros enforces mandatory access control policies that regulate inter-application communication. Privaros builds upon the basic facilities of SROS that assign identity to applications. It also leverages SROS to ensure that TLS is used for all inter-application network communication. However, it supplements the manifest-based access-control mechanism of SROS by: Ⓐ allowing end-to-end policy specifications; and Ⓑ enforcing policies within the OS.

3.3.1 End-to-end Policy Specifications

In Privaros, policies are specified in terms of permitted data flows between applications. Given a high-level policy such as ProcessLocally, BlurExportedImages or UseDroneLanes, the policy is compiled down to restrictions on inter-application communication (Section 4 will discuss policy specification in more detail). Thus, rather than require an application writer to *a priori* commit to specific topic restrictions, with Privaros, restrictions to be imposed are identified based upon the environment in which the application will execute. Consider our three example policies, for instance:

- ProcessLocally. This policy is expressed using restrictions that prevent Ⓐ any network-facing application from talking to the application that publishes the camera feed; and Ⓑ preventing the camera application from writing to the file system mounted on the SD card. A navigation application may consume the output of the camera feed, but the policy would place the same restrictions on the navigation application (*i.e.,* no network or file system communication) to prevent a leak of data from the navigation application.
- BlurExportedImages. This policy is compiled down to a restriction that all images from the camera must pass through a *blurring application* before they are consumed by a network-facing application. The blurring application is entrusted with the task of identifying and blurring out faces, car number plates, and other sensitive data.
- UseDroneLanes. This policy is compiled down to a restriction that the output of the GPS feed must pass through a *trusted logger* that stores the GPS feed in tamper-proof storage, *e.g.,* either in an audit log within the drone's hardware TEE, or in a trusted cloud server. Logs can later be analyzed to determine if the drone violated the drone lanes.

The above implementation only allows passive enforcement of UseDroneLanes, in which violations are detected *post factum*. To actively enforce the policy, a trusted application would need to analyze the GPS feed and issue navigation commands to the flight-control board to keep the drone in the lane. We restricted ourselves to the passive enforcement variant because our experimental hardware platform [80] is not integrated with a flight-control board.

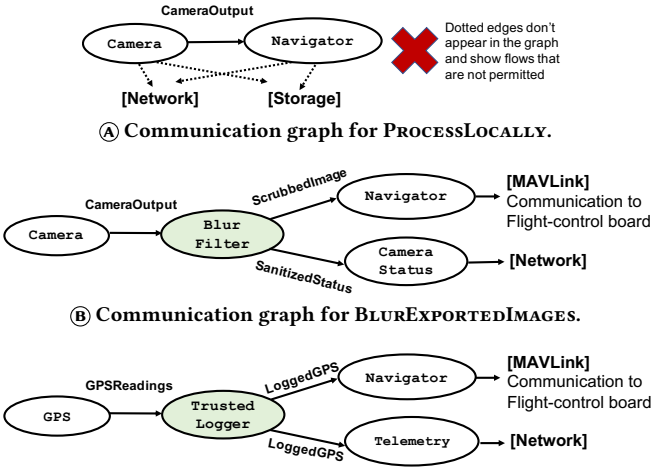


Figure 5: Illustrative communication graphs.

Figure 5 depicts the communication graph structure imposed by these restrictions. Privaros relies on *trusted applications* (*e.g.,* the blurring application and trusted logger), shown as shaded ovals in Figure 5, to permit data flows that would otherwise be forbidden. These trusted applications serve a role similar to declassifiers/endorsers from the information-flow control literature [58] or transformation procedures from the Clark-Wilson security model [19].

The host that specifies the policy must also specify any trusted applications that may be needed to enforce the policy. These trusted applications may be drawn by the host from an app store-like portal. It is the host's responsibility to ensure that the trusted applications indeed meet their privacy requirements, *e.g.,* that an app indeed identifies and blurs faces suitably. Privaros confirms to the host that the trusted applications are executing on the drone (via hardware-based attestation, Section 3.4), and ensures that data passes through these trusted applications before it reaches other downstream applications. Privaros works at the granularity of processes, and does not track how data is processed within the applications to ensure that they perform their advertised functionality (*e.g.,* blurring faces).

Note that privacy laws of the future may require drones to enforce some of these policies by default. Even with such laws in place, we foresee Privaros as being useful to hosts that wish to enforce customized policies. For example, a host may wish to specify and protect additional sensitive objects beyond those that are required to be blurred-out by law. Such hosts can use provide customized blurring applications so that BlurExportedImages can identify and blur out sensitive objects of their choosing.

3.3.2 OS-level Enforcement

Privaros restricts application-level communication within the OS (Linux, in our case) at the process level of granularity. Unlike SROS, Privaros can therefore restrict application communication via pipes, the file system, shared memory, message queues, network- and UNIX-domain sockets. Privaros validates the application binary at startup (using a digitally-signed hash of the binary) and enforces security policies on the corresponding process. As a result, Privaros's enforcement binds the application's runtime identity to its process rather than the path name of its binary (*cf.* Section 3.2, the approach used by SROS).

As is standard in many mandatory access control systems, Privaros also uses labels to enforce policies [8, 10]. Each kernel object is tagged with a label; subjects (*i.e.,* processes) also have labels. The labels of a subject and object determine whether the subject is allowed to access an object. An object's label can be changed by trusted endorsers or declassifers. This approach has been used in classical systems, *e.g.,* Bell-LaPadula [8] and Biba [10], which use centralized labels determined by a system administrator. More recent approaches that implemented information-flow tracking in modern OS kernels have used expressive decentralized label systems, where applications decide the labels they assign to their data objects [30, 46, 59, 96].

Privaros adopts a simple label system that restricts data flow between subjects using mandatory access control. Policy rules are expressed at the process-level, and determine whether a process is allowed to create/read/write to sockets, shared memory, IPC, pipes or the file system.

The approach of statically specifying which subjects can communicate with each other is somewhat more restrictive than the dynamic approach adopted by more expressive label systems (*e.g.,* [30, 46, 59, 96]). In these systems, the label associated with a data object encodes its dynamic security state, which stores the history of how it was processed (*e.g.,* its taint status). In contrast, our policies are specified as static restrictions on subject (*i.e.,* process) communication alone, and data labels do not feature in the policy specification. Thus, policy rules in Privaros must be crafted carefully to keep track of the security state of an object. This difference has practical consequences in how a policy must be expressed.

To illustrate the difference, consider enforcing the BlurExportedImages policy on a drone that has a `Navigator` application which uses images of the camera to make local navigation decisions. However, suppose that `Navigator` also needs to occasionally transmit some of these images over the network to a cloud server for further analysis (*e.g.,* if `Navigator`'s algorithms produce low confidence scores when identifying obstacles in those images). To enforce BlurExportedImages, all images sent out over the network would have to be processed by a trusted `BlurFilter` application.

With a label system that tracks the dynamic state of data objects (*e.g.,* [46, 59, 96]), the label associated with the image will determine whether it has been processed by `BlurFilter`. There are no *a priori* restrictions placed on *when* `BlurFilter` should process the image. The only restriction is that the image should be processed by `BlurFilter` at some point during its lifetime before it is sent over the network.

In contrast, in Privaros we encode BlurExportedImages by placing restrictions on application communication. One way to express this would be using the communication graph in Figure 5Ⓑ, where we place the restriction that the camera application's output can only be consumed by `BlurFilter`, whose output in turn can be consumed by `Navigator` and other applications.

However, this is clearly not the only way to express this policy and may in fact be restrictive. For example, the `Navigator` application may require a high-fidelity image stream to make decisions, and

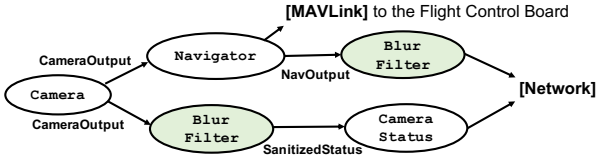


Figure 6: Alternative communication graph to enforce BlurExportedImages when `Navigator` needs a raw image feed and also has to be network-facing.

the images processed by `BlurFilter` may not be of the desired quality. In this case, the desirable option would be to use the communication graph shown in Figure 6. To realize this communication graph, Privaros could either: Ⓐ run two instances of the `BlurFilter` application (as different processes), one for each node shown in the communication graph; or Ⓑ only run one `BlurFilter` process, but modify the application to decouple the two logical flows. To process the first flow, `BlurFilter` would subscribe to CameraOutput and publish that stream after processing to SanitizedStatus. To process the second flow, it would subscribe to NavOutput and publish scrubbed images to the network. `BlurFilter` must be configured carefully to segregate these flows. Privaros must ensure that flows are directed to downstream applications correctly based on topic.

In Privaros, we chose to express policies by statically restricting subject communication to keep the design of our enforcement mechanism simple. We found empirically that this approach works in the settings we considered. But it also means that policies must be crafted carefully to balance both the host's privacy requirements and the functionality of applications executing on the drone.

We have implemented Privaros's enforcement mechanism using a kernel module. The kernel module hooks into the Linux Security Modules (LSM) framework [57] to mediate kernel operations corresponding to various communication abstractions, and enforces the access control rules specified by the host. We base our implementation on AppArmor [3], so as to leverage their policy specification language and enforcement framework, which is quite mature and stable. Applications may communicate through kernel abstractions such as pipes, files, network- and UNIX-domain sockets, shared memory, and message queues. Privaros tracks such communication by attaching the label of the sending subject with the corresponding kernel abstraction. For example, we tag files with the identity of the process that created it (using `xattrs`, extended attributes provided on modern Linux file systems), and ensure that they can be read only by the same process or other processes as allowed by the policy. In the Linux kernel, most kernel abstractions provide extra fields to store such security state.

While Privaros's in-kernel mechanisms are largely confined to the loadable kernel module, we did require some changes to the kernel itself in its networking subsystem. In particular, we found that when the LSM hook for the `sendmsg` system call is invoked, the recipient's information is not available from the `socket` data structure when the recipient is on the local host (*i.e.,* the recipient's port is not yet bound). The kernel binds this information to the socket deeper down in the network stack. Therefore, we attach the sending process identifier with the `socket` data structure, and propagate this information as the `socket` descends down the network stack into the transport layer. When the packet is processed by the kernel for delivery to the recipient process, the identities of both the sender and receiver process are available, and Privaros can decide whether the communication must be permitted.

As discussed earlier, ROS supports distributed robotics platforms, where the publisher and subscriber need not be on the same host. Thus, for instance, ROS can support fleets of drones where an application on one drone publishes data that can be consumed by applications running on other drones. Thus, network packets may leave the drone as well. One could consider a situation where a fleet of drones enforces BlurExportedImages by running the `BlurFilter` application on just one drone (say, the fleet coordinator drone), and only allowing outbound network communications (*i.e.,* out of the fleet) from that drone. In such cases, simply forbidding network packets containing the raw image feed from leaving a drone would be too restrictive. Instead, network packets must be allowed to the fleet coordinator, but not to other servers. Privaros uses a whitelist of allowed domains (*e.g.,* as done in Weir [59] and Hails [36]) to allow such communication. The LSM hook for `sendmsg` determines whether the packet will leave the localhost, and if so, allows the communication only if the IP address of the destination (*e.g.,* the IP address of the fleet coordinator) appears in a whitelist.

3.3.3 Modifications to ROS

A key problem arises when Privaros attempts to enforce policies such as BlurExportedImages or UseDroneLanes with off-the-shelf ROS applications. These policies require redirection of flows through trusted declassifiers before they can be consumed by downstream applications. However, the manifests of ROS applications will likely not allow redirection to happen easily. For example, consider a camera application's manifest that allows it to publish to the topic CamOutput and a `Navigator` application whose manifest declares that it subscribes to CamOutput. We cannot simply introduce a trusted `BlurFilter` application between the `Navigator` and the camera applications. Privaros's OS-level mechanisms will permit the information flow from the camera process to the `BlurFilter` process and the output of the `BlurFilter` process to be consumed by `Navigator`, based on the policy. However, the ROS publish/subscribe system will not set up the flow because the topics do not match (`BlurFilter` publishes to ScrubbedImage, to which `Navigator` has not subscribed).

One way to address this problem is to generate manifest files for ROS applications based on the whitelisted flows in the communication graph. For example, `BlurFilter`'s manifest would declare that it subscribes to the topic CamOutput and publishes to the topic ScrubbedImage. In turn, `Navigator`'s manifest would allow it to subscribe to ScrubbedImage (but not to CamOutput). However, this approach may not be practical for off-the-shelf ROS applications whose manifests are part of their identity (*i.e.,* X.509 certificates). The key difficulty is that fresh X.509 certificates have to be issued for each manifest configuration, which may not be feasible.

Privaros modifies ROS to allow flows to be transparently redirected between applications, as requested in the policy. In particular, it modifies the publish/subscribe system in ROS to: Ⓐ tear down an existing communication channel between a pair of applications; Ⓑ setup a new connection between applications, thereby allowing us to introduce a trusted declassifier; and Ⓒ assign a ROS topic and

type to each newly-established connection. Recall from Section 3.2 that application manifests only specifies the topic; the type is only available from the ROS runtime. Privaros probes the publish/subscribe system to identify the type, and uses this information to annotate newly-added edges in the communication graph.

Note that these modifications are required only to *enable* communication between processes that is already permitted by the MAC-based enforcement of the OS. Privaros relies solely on OS-level mechanisms to prevent applications from communicating. Thus, the modifications to ROS are *not* part of our TCB. In particular, Privaros allows a pair of applications to communicate only if allowed by both ROS and the OS's MAC-based policy enforcement.

Privaros works with off-the-shelf ROS applications. Recall from Figure 3 that applications use the ROS API via the ROS library that is linked into the process address space. The modifications discussed above are implemented within the ROS library and are transparent to ROS applications, which dynamically link against the ROS library on the drone platform. Privaros's kernel-level mechanisms are also transparent to ROS applications.

3.4 Role of the Hardware TEE

As previously discussed, we use a TEE based on ARM TrustZone in our prototype implementation. Our prototype makes use of the TEE in the standard way for attestation [75]. A TrustZone processor offers two *worlds* of execution. The *normal world* executes untrusted applications and is typically the environment with which the end-user interacts. In our case, all drone applications, and Privaros (*i.e.,* ROS and the OS-based mechanisms) run in the normal world.

The *secure world* manages the drone's private key, implements remote attestation, and is therefore trusted and protected by secure boot. Its memory is isolated from the normal world. After booting securely, the secure world boots the normal world. It obtains and stores integrity measurements of the normal world boot process (*i.e.,* a hash chain of software initialized during the boot sequence). These measurements can be used in remote attestations to convince a challenger (*e.g.,* the aviation authority or any host airspace) that the normal world booted with an untampered TCB. The attestation report also includes a log of the applications started by Privaros (as in TPM-based integrity measurement of applications executed over the system lifetime [75]). Hosts can use this log to verify that any trusted declassifier applications that they entrust for policy enforcement in Privaros are running on the drone.

Standard TEE-based attestation can detect attempts by a malicious DSP to install certain kinds of rootkits in the normal world. Rootkits that modify the normal world's kernel code or static data can be detected using integrity measurements at boot time. Although not currently implemented in our prototype, prior work has developed TEE-based methods to protect the normal world from advanced rookits, *e.g.,* those that use direct kernel object manipulation. These methods have primarily been developed to offer real-time protection for kernel code [7, 35] or obtain runtime snapshots of the normal world memory for analysis [79]. In addition, a CFI-protected [34] normal world kernel (attested at boot-time using standard TEE-based integrity measurement) can provide real-time protection from various attacks directed against the kernel. We plan to integrate these methods in our prototype in future work.

4 POLICY INTERFACE

Specifying and loading policies. Policies in Privaros are specified using communication graphs. The graph identifies a whitelist of permitted flows between applications. Edges in the graph may be annotated with a topic name to denote the ROS topic that restricts the communication between that pair of applications to that topic alone. Edges may lack an annotation if applications are allowed to communicate outside the purview of ROS, *e.g.,* using OS primitives.

Security administrators specify these policies by hand. However, a real-world drone may run dozens of ROS applications in addition to tens of daemons or other processes running natively on top of the underlying OS. For example, the communication graph on our experimental platform (Section 5) has 29 nodes and 69 edges, even without any ROS applications running on it. Writing a comprehensive whitelist of allowed flows would therefore be time-consuming and might erroneously omit certain flows that prevent applications from working. We thus built a tool to extract communication graphs from a running drone (encapsulating all the flows between applications on that drone), which the security administrator can then use as a starting point and refine. We view this approach as being similar to the popular practice of using the `audit2allow` tool to write SELinux and SEAndroid policies. While we also fully acknowledge the usability concerns with `audit2allow`, we view policy specification as an orthogonal problem that must be studied separately. Advanced policy analysis tools, such as those developed to configure SEAndroid policies [17, 85, 86], could be brought to bear as better alternatives to formulating policies.

Once a policy is written, it can be loaded into the drone for enforcement by Privaros. We have built a user-level agent that identifies the process IDs of applications running on the drone, and translates the application names in the policy to the corresponding process IDs. Privaros then applies the constraints imposed by this whitelist policy directly on the processes. The policy itself is expressed as a user-space file, but is serialized and loaded into the kernel via a user-agent (similar to the infrastructure provided by AppArmor, which Privaros builds upon). The policy can be updated at any time by simply unloading the old policy and loading a new one, without restarting any applications. Privaros thus transparently supports dynamic policy updates. This feature is important because dynamic policy updates may be required as the drone moves from one host airspace to another.

On integration with India's Digital Sky portal. India's Digital Sky portal offers a Web-based service [37] via which drone operators indicate the proposed delivery route using a visual map-based interface. The Digital Sky server permits the delivery run if the route does not intersect any no-fly or other restricted zones.

We can extend the same interface for the setting that Privaros considers. Each host specifies their privacy policies and geo-tags the policy with the coordinates of their airspace. The Digital Sky server stores a database of all registered hosts and their policies.

When a drone operator uses the Digital Sky server to mark the delivery route, the server identifies all host airspaces that the route intersects with (recall Figure 1). It then sends all the associated policies to the drone, where they are stored in the drone's local storage. The policies can be communicated to the drone using the

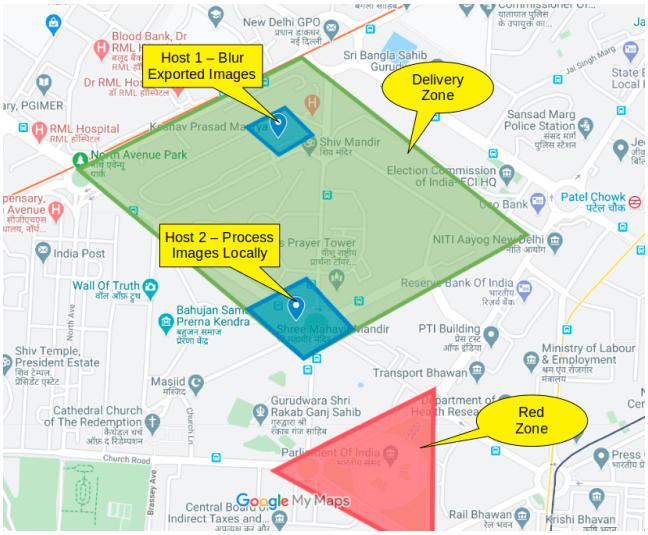


Figure 7: Integrating policy specification in Privaros with the Digital Sky interface.

same infrastructure (be it WiFi, 5G, or LTE) that the Digital Sky server uses to send NPNT approval certificates to the drone, prior to take-off. For the case of delivery drones, this step can happen at the warehouse from which the delivery run starts, where the availability of WiFi or wired network infrastructure can be assumed. Once the drone is airbrone, Privaros continuously monitors the GPS coordinates of the drone, determines if it is entering a host airspace, and loads the corresponding policy from local storage for enforcement. It unloads the policy as it departs that host's airspace.

We obtained the code of the Digital Sky Web server [27, 28] and created a mock setup in our lab. Figure 7 shows a screenshot of the Web server interface in which a drone operator has declared a drone's proposed delivery zone. It intersects two host airspaces, who have declared their privacy requirements. Figure 7 also shows a red-zone (in this case, the Indian Parliament house in New Delhi; such sites would be identified by the aviation authority) that this drone's proposed delivery zone avoids.

Using the Digital Sky portal also has the benefit of simplifying the UI that a host would use for policy specification. Recall that Privaros policies are specified as a communication graph of whitelisted flows. The key challenge in deploying this approach is that the communication graph must be customized for *each policy* and *each drone*. For example, to write the policy specification for ProcessLocally for *a particular drone*, the host would have to Ⓐ identify all network-facing applications on that drone; and Ⓑ carefully create a communication graph in which the camera application never talks to a network-facing application. This exercise would have to be repeated for BlurExportedImages, UseDroneLanes and other policies of interest. And the whole exercise has to be repeated for every drone that potentially enters that host's airspace.

Digital Sky simplifies this exercise because it contains a database of all registered drones. This database could simply be extended to maintain a list of all applications installed on the drone, which Digital Sky can reliably obtain from the drone using the hardware TEE. We could pre-compute the communication graphs for various popular policy choices (*e.g.*, ProcessLocally, BlurExportedImages, UseDroneLanes) in an offline fashion and store them in the database. From the host's perspective, the UI to specify policies can be simplified to a pull-down menu of common policy choices that they may wish to apply to their airspace. When a drone expresses its intent to fly to the host's airspace, the Digital Sky server looks up the database to obtain the pre-computed communication graph corresponding to the combination of that drone and policy, and sends it to the drone.

5 EVALUATION

We implemented Privaros on a system running Ubuntu 18.04 with Linux kernel version 4.9. We used ROS version 2 (Dashing Diademata) with eProsima FastRTPS version 1.8.2 [31, 32] as the underlying implementation of the DDS protocol. We enhanced it with the Secure ROS module available for ROS version 2 to enable TLS communication and to leverage the SROS application manifest infrastructure. We used AppArmor's user-space policy specification framework (version 2.13) to specify and download policies into the kernel for enforcement. Overall, we added or modified 402 lines of code in the ROS client library for C++ and 1651 in Linux to implement Privaros (measured with `sloccount`). We also modified 431 lines in the AppArmor user-space tool to parse policies, and added 213 lines of Python/bash code to support redirection of flows between ROS applications.

We evaluated Privaros on an Nvidia Jetson TX2 [80] development kit, with a dual-core Denver 2 64-bit CPU and quad-core ARM A57 complex, 8 GB LPDDR4 memory and 32 GB eMMC flash storage. Our choice of Jetson was motivated by the fact that unlike most off-the-shelf drones, it is equipped with a hardware TEE (based on ARM TrustZone) and allows programmable access to both the secure world and the normal world. The specification of the Jetson board is similar in architecture to the companion boards of commercially-available drones. It also consists of 256 Nvidia CUDA cores, making it the companion board of choice for navigation software that makes extensive use of graphics processing units, *e.g.*, those that use deep-learning based navigation. We reflashed the normal world of this board with a Privaros-enhanced software stack.

Our evaluation considers two questions: Ⓐ How effective is Privaros at enforcing policies, and how secure is it in comparison to SROS? (Section 5.1); Ⓑ What is the impact of Privaros's mechanisms on latency, CPU utilization and power consumption, as evaluated with microbenchmarks? What is the impact of redirecting communication through trusted applications? (Section 5.2).

5.1 Robustness of Policy Enforcement

To showcase that Privaros offers defense-in-depth, we built a malicious application that SROS cannot confine, and demonstrate the multiple layers Privaros provides to confine this application. Consider a `Camera` application that publishes to a topic called CameraOutput. The application publishes two types of data under this topic: Ⓐ the image feed from the camera under type CameraOutput::ImageType, and Ⓑ its status, under type CameraOutput::StatusType (see Figure 8Ⓐ).

The primary goal of publishing CameraOutput::StatusType is so that it can be consumed by `CameraStatus`, which is a benign

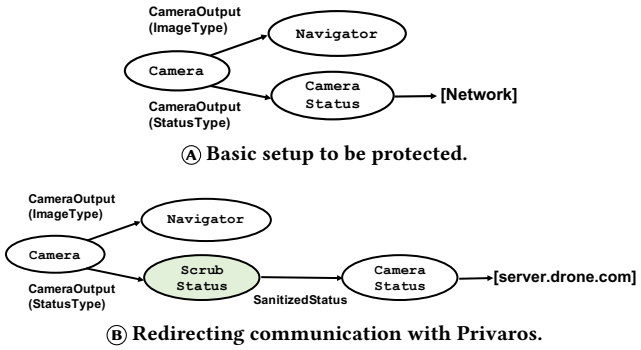


Figure 8: Setup used in the experiments to demonstrate robustness of policy enforcement in Privaros (Section 5.1).

application that subscribes to the topic CameraOutput only to read the data published under the type CameraOutput::StatusType. This application periodically uploads the camera's operating status to the drone operator's server that monitors the health of its fleet.

We examine various ways in which it is possible for an attacker to write a malicious application called `BadCameraStatus` that subscribes to the topic CameraOutput but instead reads CameraOutput::ImageType and transmits it over the network. The primary concern of a host would be to ensure the privacy of their image feed. We now examine how SROS and Privaros compare in their ability to prevent the camera's output from being leaked either accidentally, or through malicious applications such as `BadCameraStatus`. We empirically validated each of the following attacks by implementing them and showing that Privaros prevents them:

• **Certificate checks.** The certificate checks in SROS can prevent an overt attempt at an attack, such as an attacker attempting to install `BadCameraStatus`. Under the assumption that such an application will not receive a valid certificate from a trusted authority, SROS certificate validation would fail, and SROS would not install the application. SROS would also prevent such an application (assuming it got installed) from subscribing to CameraOutput if this is not declared in its manifest. For context, ROS (without SROS) would simply allow `BadCameraStatus` to be installed and allow it to subscribe to CameraOutput and even publish messages to the same topic (*e.g.,* a fake image feed).

However, the checks performed by SROS can easily be bypassed. An attacker (*e.g.,* a malicious drone administrator) could replace the binary of the benign `CameraStatus` application with that of `BadCameraStatus` at the same file path (*cf.* Section 3.2). The attacker would launch this program using the same file path as the benign `CameraStatus` application, but it would perform the functionality intended by `BadCameraStatus`. SROS checks X.509 certificates of apps, but does not associate the application's identity with their binary and instead only their full path name, and would therefore miss this attack. Privaros prevents this attack because it checks the application binary's hash during certificate validation.

• **Redirection of app communication.** To prevent accidental disclosure of the `Camera` application's image feed, a host could require that no network-facing application directly subscribe to CameraOutput. It could instead require the camera's status to pass through a trusted application called `ScrubStatus`, which performs sanity-checks on the status feed. For example, `ScrubStatus` could ensure that the status feed only transmits a single byte. It could also rate-limit the flow (*e.g.,* status updates allowed only once every 10 seconds), thereby mitigating the effects of any side-channels, via which an attacker attempts to leak images via the status feed, byte-by-byte.

One way to implement such enforcement in SROS would be to cleanly decouple the topics representing the image feed and the status feed by having the `Camera` application publish to two topics, ImageFeed and StatusFeed (because SROS only matches topics, and not types, as discussed in Section 3.2). The `ScrubStatus` application could subscribe to StatusFeed, but not ImageFeed, and then publish the output to a topic SanitizedStatus to which `CameraStatus` could subscribe and transmit over the network.

Privaros can enforce this policy even if it is not easily possible to decouple the topics, *e.g.,* because the `Camera` application code is not available or its manifest cannot be rewritten. With Privaros, the trusted `ScrubStatus` application could still subscribe to CameraOutput, but only read the CameraOutput::StatusFeed type, and publish to SanitizedStatus (see Figure 8Ⓑ). Note that Privaros's modifications to ROS (Section 3.3.3) are essential to allow `CameraStatus` to read the output of the `ScrubStatus` application. This is because the manifest of `CameraStatus` only allows it to subscribe to the topic CameraOutput and not to SanitizedStatus. However, Privaros's modifications to ROS allow `CameraStatus` and `ScrubStatus` to communicate with each other.

• **Direct communication via OS.** The `BadCameraStatus` application could directly establish an inter-process channel (say, via UNIX domain sockets) to communicate with the `Camera` application, obtain images and send it over the network. SROS cannot mediate non-publish/subscribe communication and would allow this attack. The OS-level mechanisms of Privaros prevent any communication between the processes unless allowed by the policy. Assuming the application redirection discussed above (through `ScrubStatus`), Privaros can prevent any form of direct communication between `Camera` and `BadCameraStatus` (or even `CameraStatus`). All communication to network facing applications would have to go through the process that implements `ScrubStatus`.

• **Whitelisting network domains.** Finally, `CameraStatus` is a network-facing application. Privaros uses whitelisting can ensure that the output of `CameraStatus` only goes to a particular IP address. SROS does not confine network communication this way.

5.2 Performance

We used microbenchmarks to measure the impact of Privaros's core mechanisms on latency, CPU utilization, and power consumption. We used `PerformanceTest` [2], a DDS microbenchmark from Apex AI that is designed to evaluate the performance of publish/subscribe systems. `PerformanceTest` consists of a suite of workloads, each of which runs publishers and subscribers in different threads, and measures the latency involved in publishing/subscribing. Figure 9 presents the details of the `PerformanceTest` workloads we used. We ran each workload configured to use one publisher and one subscriber, publishing at a rate of 10Hz for a duration of 10 seconds.

Workload	Type of data published/subscribed
`Array`	Simple byte array
`PointCloud`	Collection of N-dimensional points (*e.g.,* 2D images produced by camera depth sensors)
`Struct`	Structure holding a set of bytes (*e.g.,* 16 bytes in `Struct16`)
`NavSat`	Status of navigation satellite
`Range`	Single range reading obtained from a range sensor

Figure 9: Workloads from `PerformanceTest` [2].

Workload	Latency (ms)	CPU (%)	Power (mW)
Baseline			
`Array1m`	16.255	6.728	2435.133
`PointCloud1m`	16.160	6.612	2441.062
`Struct32k`	6.494	2.526	2225.375
`NavSat`	1.543	1.381	2349.353
`Range`	1.433	1.378	2268.059
Privaros			
`Array1m`	17.225 (+5.9%)	7.050 (+4.8%)	2508.222 (+3.0%)
`PointCloud1m`	17.386 (+7.6%)	7.141 (+8.0%)	2437.294 (-0.2%)
`Struct32k`	7.109 (+9.5%)	2.665 (+5.5%)	2500.412 (+12.4%)
`NavSat`	1.922 (+24.6%)	1.506 (+9.1%)	2389.167 (+1.7%)
`Range`	1.928 (+34.5%)	1.501 (+8.9%)	2367.412 (+4.4%)

Figure 10: Microbenchmark performance.

Workload	Baseline Latency (μs)	Privaros Latency (μs)
Pipe	15.471	15.640 (+1.093%)
UNIX domain sockets (TCP)	20.015	23.188 (+15.9%)
UDP (localhost)	35.039	35.374 (+1.0%)
TCP (localhost)	38.473	38.764 (+0.8%)
UDP (RPC)	51.549	52.335 (+1.5%)
TCP (RPC)	49.457	49.977 (+1.1%)

Figure 11: Experiments using `lmbench`.

`PerformanceTest` reports the latency numbers for each workload. We measured the CPU utilization as the workload ran. To measure power consumption, we used the 3-channel INA3221 hardware power monitors on the Jetson TX2, which reports power draw of the board in milliwatts. Figure 10 reports the results of our experiments. The baseline column reports the results of performing these experiments on a vanilla ROS/Linux setup with SROS enabled, and serves as the baseline. The Privaros column reports the same numbers with the workloads running on Privaros. As these numbers show, Privaros imposes only a marginal increase in latency (under 10% except when the latency numbers themselves are under 2ms) and power draw (under 5% except in the case of `Struct32k`). Because drones are battery-powered, with current drones only providing an average flight time of about 20 minutes on a single charge, it is critical for Privaros to be efficient with respect to power draw. Finally, we also measured the performance impact imposed by Privaros's hooks on individual kernel network subsystems using the `lmbench` [55] benchmark. Figure 11 reports these results.

Finally, we studied the performance impact of redirecting data flow through a trusted application, as would be required for example to enforce BlurExportedImages or UseDroneLanes. Since these trusted applications are now part of the data-flow path, their presence will likely increase the latency of data delivery and overall power consumption. To illustrate the impact of trusted applications, we use the example of a `Camera` application whose images must pass through a trusted `BlurFilter` application before they are consumed by downstream applications. We measure the baseline performance, without application redirection, and two variants of the `BlurFilter` application: Ⓐ a null-filter that simply redirects network flows but does not otherwise process the image (to measure the raw cost of redirection), and Ⓑ a second one that is based on OpenCV, and blurs all image frames by 10%. In this case, the application's processing logic itself performs non-trivial image-processing and consumes CPU and power.

Scenario	Latency (ms)	Power (mW)
No redirection	8.124	4749.400
`BlurFilter/Null`	17.509 (+115.5%)	4836.200 (+1.8%)
`BlurFilter/OpenCV`	21.511 (+164.8%)	5132.400 (+8.1%)

Figure 12: Performance impact of flow redirection.

Figure 12 reports the results of this experiment. The end-to-end latency of transmitting images from the camera to the network increases significantly when the `BlurFilter` application is introduced. The increase in latency is as expected, because of the additional user-space element involved in the outbound network path, and the associated transitions of the data packets between kernel-space and user-space. The end-to-end increase in power consumption remains under 10% even with OpenCV-based blurring.

Power consumption overheads will depend on the nature of processing involved in the trusted application. A real-world drone running dozens of applications will require many such trusted declassifiers, depending on the host policy to be enforced. Communication graphs must be carefully configured to minimize the number of distinct trusted elements required and their power consumption.

6 RELATED WORK

Drones and privacy. To our knowledge, there has not been much prior work focusing on enforcing privacy policies in drones. In a prior paper [83], we proposed the vision of restricted airpsaces for drones, in which host-specified policies would be enforced on guest drones. This paper builds upon that vision but makes significant additional contributions. In particular, this paper fully explores the challenges of building an enforcement framework, which was only sketched in our vision paper. We learned that it was not possible to build an enforcement system on top of ROS alone, as outlined in our prior paper. We showed that even the primitives provided by SROS are insufficient to prevent a number of attacks and that OS-level enforcement is central to ensuring robust policy enforcement. Finally, in this paper, we also showed how our policy-enforcement framework can be integrated with Digital Sky.

Nassi *et al.* [60] consider the problem of determining whether a drone's first-person view violates an individual's privacy. A first-person view projects the drone's camera feed as to a ground-based remote controller, operated by a human(for purposes of navigation). The communication between the remote controller and the drone is encrypted. Nassi *et al.* develop a cryptanalysis technique by which an analyst with access to the encrypted first-person view feed can determine if the feed is focused on a particular object(or person) of interest. They apply physical perturbations to the object, *e.g.,* by

shining a light on it, at chosen points in time. The cryptanalysis determines if the encrypted feed is affected by these perturbations; if yes, they determine that the camera is focused on the object. They also develop techniques that spatially localize the offending drone by analyzing the first-person view feed.

Drones and security. In contrast to privacy, there is much prior work on security of drones. These range from using hardware TEEs to ensure that applications running on drones are able to securely access sensor data from its peripherals [47] and to ensure that drones only fly along drone lanes [48] (*i.e.,* UseDroneLanes), to investigating attacks against and protecting drones from common vulnerabilities, such as cleartext communication between drones, signal jamming and GPS spoofing [41, 43, 44, 65, 77, 78, 82, 87]. Privaros can benefit from techniques developed to defend against these attacks but has an orthogonal focus on enforcing host-specified privacy policies on drones.

The ROS community has also actively identified security vulnerabilities and attacks that stem from the unauthenticated, plaintext, publish/subscribe-based communication in ROS [25, 52]. There have been proposals to use encrypted communication between applications [69], and to integrate TLS with the core libraries of ROS [25, 26]. SROS [90–92], which is under active development, incorporates many of these ideas. As already discussed, Privaros enhances the basic security features of SROS, eliminates some of its key shortcomings, and adds the ability to enforce privacy policies.

While the above projects focus on securing drones from attacks, there is also work on detecting drones *i.e.,* securing physical premises against unauthorized rogue drones. These include methods to detect drones using their radar [33] or radio-frequency signature [11, 61], computer vision techniques to identify drones [73], acoustic arrays that detect the sound of the drone's motors [14, 15], and hybrid combinations of these techniques [81]. These techniques are undoubtedly important in formulating regulations to operate drones. However, they are orthogonal to Privaros whose focus is on ensuring that authorized and legally-permitted drones conform to the privacy requirements of a host airspace.

Mandatory and context-based access control. The idea of controlling the flow of information in computer systems can be traced back to some of the classic papers in computer security [8, 10, 19, 53]. SELinux [50], SEAndroid [76], and related systems (*e.g.,* [13, 63]) have brought to bear some of these methods to modern OS settings. In these systems, subject and object labels are set by the system administrator, and the enforcement system applies label flow rules.

In modern device-centric settings, some of these concepts have been adapted as context-based access control systems [12, 13, 23, 56, 70, 88], where the context in which the device is used (*e.g.,* at home or in the workplace) determines the policies that must be enforced on the device. Some of these systems [13, 56, 70] also employ methods to actively infer the context in which the device is being used, and trigger the enforcement of the appropriate policy. Privaros can also be viewed as a context-based access control system for delivery drones, and our focus in this paper has primarily been on building an enforcement mechanism integrated with ROS. Key contributions of this paper include: Ⓐ exploring the shortcomings of SROS; Ⓑ designing and implementing the cross-stack changes required for policy enforcement; and Ⓒ redirecting flows through trusted applications for policy compliance. Privaros's policies are location-tagged, and the drone's GPS coordinates serve as the policy trigger to load access control policies. As discussed in Section 4, Privaros allows dynamic loading/unloading of policies as a guest drone navigates between host airspaces.

Information-flow control. Recent attempts to enforce information flow control on operating systems [46, 96] and Android [30, 59, 93] have focused on *decentralized* information-flow control (DIFC) [58]. DIFC systems differ from classic systems in that each application can specify its own labels, and the role of the system is to only use these labels to enforce certain rules on information flow. DIFC is particularly well-suited for settings where each application wants to control how *its own data* is used by the rest of the system. In contrast, our setting requires us to apply host-specified privacy policies uniformly to all applications on the drone. Privaros is therefore closer in spirit to the earlier work on using mandatory access control to regulate information flow [8, 10, 19], SELinux [50] and SEAndroid [76]. Privaros adapts these concepts to a ROS-based platform and tightly integrate ROS-level and OS-level mechanisms.

7 CONCLUSIONS

In this paper, we presented Privaros, a framework that enforces privacy policies specified on guest drones visiting host airspaces for delivery runs. Our main conclusions are that:

- The problem of enforcing host-specified privacy policies on guest drones can be modeled as one of controlling the flow of data between applications executing on the drone;
- Existing mechanisms in ROS do not suffice to enforce these kinds of policies. Tight integration of ROS-level and OS-level mechanisms, as provided in Privaros, are necessary for robust enforcement;
- Policy specification for Privaros can be integrated with upcoming drone regulatory platforms such as Digital Sky;
- The core mechanisms of Privaros impose low overheads on latency and power consumption. However, the host's policies may require the drone to execute trusted applications, which may themselves impose additional latency or consume additional power.

Supplementary material. Additional material related to this paper, including slides and code may be obtained from http://www.csa.iisc.ac.in/~vg/papers/ccs2020.

ACKNOWLEDGMENTS

We thank: Ⓐ John Johansen and Seth Arnold from the AppArmor development team for helping us resolve technical difficulties during the development of Privaros; Ⓑ Ruffin White for his insights on SROS, for illustrating the shortcomings of ROS1, and motivating us to develop Privaros atop ROS2; Ⓒ Earlence Fernandes for shepherding the paper and the reviewers from CCS'20 for their comments; Ⓓ Robert Bosch Centre for Cyber-Physical Systems, IISc Bangalore, and a Ramanujan Fellowship from the Government for India, for generous grants that funded this work. The last author dedicates this paper to the memory of his mentor and friend, Liviu Iftode of Rutgers University.

REFERENCES

[1] United States Federal Aviation Administration. FAA Aerospace Forecast: Fiscal Years 2019-2039, 2019. https://www.faa.gov/data_research/aviation/aerospace_forecasts/media/FY2019-39_FAA_Aerospace_Forecast.pdf.

[2] Apex AI. Performance Test. https://gitlab.com/ApexAI/performance_test/.

[3] AppArmor–an effective and easy-to-use Linux application security system. https://gitlab.com/apparmor/apparmor/wikis/home/.

[4] Ardupilot Autopilot Software. https://ardupilot.org/.

[5] ARM. Security technology building a secure system using TrustZone technology (white paper). *ARM Limited*, 2009. https://static.docs.arm.com/genc009492/c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[6] United Kingdom-Civil Aviation Authority. The Drone and Model Aircraft Code—Protecting people's privacy. https://register-drones.caa.co.uk/drone-code/protecting-peoples-privacy.

[7] A. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world. In *ACM Conf. on Computer and Communications Security*, 2014.

[8] D. E. Bell and L. LaPadula. Secure computer system: Unified exposition and Multics interpretation, March 1976. Tech. Report MTR-2997, MITRE Corporation.

[9] W. C. Bennett. Civilian drones, privacy, and the federal-state balance. In *The Brookings Institution—The Project on Civilian Robots*, September 2014. https://www.brookings.edu/research/civilian-drones-privacy-and-the-federal-state-balance/.

[10] K. J. Biba. Integrity considerations for secure computer systems, June 1975. Technical Report MTR-3153, MITRE Corporation.

[11] S. Birnbach, R. Baker, and I. Martinovic. Wi-fly?: Detecting privacy invasion attacks by consumer drones. In *Network and Distributed Systems Security Symposium*, 2017.

[12] F. Brasser, D. Kim, C. Liebchen, V. Ganapathy, L. Iftode, and A.-R. Sadeghi. Regulating ARM TrustZone devices in restricted spaces. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.

[13] S. Bugiel, S. Heuser, and A-R. Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *USENIX Security*, 2013.

[14] J. Busset, F. Perrodin, P. Wellig, B. Ott, K. Heutschi, T. Rühl, and T. Nussbaumer. Detection and tracking of drones using advanced acoustic cameras. *Unmanned/Unattended Sensors and Sensor Networks XI and Advanced Free-Space Optical Communication Techniques and Applications*, 2015.

[15] E. E. Case, A. M. Zelnio, and B. D. Rigling. Low-cost acoustic array for small UAV detection and tracking. In *IEEE Natl. Aerospace & Electronics Conf.*, 2008.

[16] V. Chang, P. Chundury, and M. Chetty. "Spiders in the sky": User perceptions of drones, privacy, and security. In *ACM SIGCHI Conference on Human Factors in Computing Systems*, 2017.

[17] H. Chen, N. Li, W. Enck, Y. Aafer, and X. Zhang. Analysis of SEAndroid policies: Combining MAC and DAC in Android. In *Annual Computer Security Applications Conference*, 2017.

[18] ChibiOS free embedded RTOS. http://chibios.org.

[19] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Research in Security and Privacy*, 1987.

[20] CNBC. Amazon says this business opportunity could make you up to $300k a year—here's how to get into the program, September 2018. https://www.cnbc.com/2018/09/06/amazon-delivery-service-partner-program-gets-thousands-of-applications.html.

[21] British Broadcasting Corporation. Big rise in drone jail smuggling incidents, February 2016. http://www.bbc.com/news/uk-35641453.

[22] British Broadcasting Corporation. Gatwick airport: Drones ground flights, 2018. https://www.bbc.com/news/uk-england-sussex-46623754.

[23] M. J. Covington, P. Fogla, Z. Zhan, and M. Ahamad. A context-aware security architecture for emerging applications. In *Annual Computer Security Applications Conference*, 2002.

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

[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 *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 4477–4482. IEEE, 2016.

[27] Digital Sky Web Server back-end. http://github.com/iSPIRT/digital-sky-api.

[28] Digital Sky Web Server front-end. http://github.com/iSPIRT/digital-sky-app.

[29] How To Get Flipkart Franchise. https://www.steptowardbusiness.com/flipkart-franchise.

[30] W. Enck, P. Gilbert, B-C. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *ACM/USENIX Symposium on Operating System Design and Implementation*, 2010.

[31] The most complete DDS by eprosima, the middleware experts (https://www.eprosima.com). https://github.com/eProsima/Fast-RTPS.

[32] eProsima fast DDS docs. https://fast-rtps.docs.eprosima.com/en/latest/.

[33] T. Eshel. Mobile radar optimized to detect UAVs, precision guided weapons. *Defense Update*, 2013.

[34] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *IEEE European Symposium on Security and Privacy*, 2016.

[35] X. Ge, H. Vijayakumar, and T. Jaeger. Sprobes: Enforcing Kernel Code Integrity on the TrustZone. In *IEEE Workshop on Mobile Security Technologies*, 2014.

[36] D. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazieres, J. Mitchell, and A. Russo. Hails: Protecting Data Privacy in Untrusted Web Applications. In *USENIX Symposium on Operating System Design and Implementation*, 2012.

[37] Government of India. Digital Sky Portal — Office of the Director General of Civil Aviation. https://digitalsky.dgca.gov.in/.

[38] Government of India. Office of the Director General of Civil Aviation: DGCA RPAS Guidance Manual, First Edition, November 2018. https://diceindia.org.in/wp-content/uploads/DGCA-RPAS-Guidance-Manual.pdf.

[39] Government of India. Office of the Director General of Civil Aviation: DGCA RPAS Guidance Manual, Revision One of First Edition, June 2019. https://diceindia.org.in/wp-content/uploads/Updated-DGCA-RPAS-Guidance-Manual.pdf.

[40] Aviation Systems Group. Drones and the UK Privacy Law. https://asg.ltd/drones-and-privacy-law/.

[41] K. Hartmann and C. Steup. The vulnerability of UAVs to cyber attacks-an approach to the risk assessment. In *IEEE Intl. Conference on Cyber Conflict*, 2013.

[42] T. Jain and G. Cooperman. DMTCP: Fixing the single point of failure of the ros master. In *ROSCON 2017: the ROS Developers Conference*, 2017.

[43] A. Y. Javaid, W. Sun, V. K. Devabhaktuni, and M. Alam. Cyber security threat analysis and modeling of an unmanned aerial vehicle system. In *IEEE Conference on Technology for Homeland Security*, 2012.

[44] A. J. Kerns, D. P. Shepard, J. A. Bhatti, and T. E. Humphreys. Unmanned aircraft capture and control via GPS spoofing. *Journal of Field Robotics*, 31(4), 2014.

[45] C. Koettl and B. Marcolini. A closer look at the drone attack on Maduro in Venezuela, August 2018. https://www.nytimes.com/2018/08/10/world/americas/venezuela-video-analysis.html.

[46] 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.

[47] R. Liu and M. Srivastava. PROTC: Protecting drone's peripherals through ARM TrustZone. In *3rd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications*, 2017.

[48] T. Liu, A. Hojjati, A. Bates, and K. Nahrstedt. Alidrone: Enabling trustworthy proof-of-alibi for commercial drone compliance. In *IEEE 38th International Conference on Distributed Computing Systems*, 2018.

[49] Amazon Logistics. Amazon Logistics—Delivery Services Partners Program. https://logistics.amazon.com/marketing/opportunity.

[50] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *USENIX Annual Technical Conference*, 2001.

[51] MAVLink Developer Guide. https://mavlink.io/en/.

[52] J. McClean, C. Stull, C. Farrar, and D. Mascareñas. A Preliminary Cyber-Physical Security Assessment of the Robot Operating System (ROS). In *Unmanned Systems Technology XV*, volume 8741, page 874110. International Society for Optics and Photonics, 2013.

[53] M. D. McIlroy and J. A. Reeds. Multilevel security in the UNIX tradition. *Software–Practice and Experience*, 22(8), 1992.

[54] G. McNeal. Drones and aerial surveillance: Considerations for legislatures. In *The Brookings Institution—The Project on Civilian Robots*, November 2014. https://www.brookings.edu/research/drones-and-aerial-surveillance-considerations-for-legislatures/.

[55] L. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *1996 Annual Conference on USENIX Annual Technical Conference*, 1996.

[56] M. Miettinen, S. Heuser, W. Kronz, A.-R. Sadeghi, and N. Asokan. Conxsense—context profiling and classification for context-aware access control. In *ACM Symposium on Information, Computer and Communications Security*, 2014.

[57] J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *USENIX Security*, 2002.

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

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

[60] B. Nassi, R. Ben-Netanel, A. Shamir, and Y. Elovici. Drones' Cryptanalysis – Smashing Cryptography with a Flicker. In *IEEE Symposium on Security and Privacy*, 2019.

[61] P. Nguyen, H. Truong, M. Ravindranathan, A. Nguyen, R. Han, and T. Vu. Matthan: Drone presence detection by identifying physical signatures in the drone's RF communication. In *15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017.

[62] Office of the Australian Information Commissioner. Surveillence and monitoring–drones. https://www.oaic.gov.au/privacy/your-privacy-rights/surveillance-and-monitoring/drones/.

[63] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *Annual Computer Security Applications Conference*, 2009.

[64] Pixhawk 4 Flight Controller. https://docs.px4.io/v1.9.0/en/flight_controller/pixhawk4.html.
[65] J-S. Pleban, R. Band, and R. Creutzburg. Hacking and securing the AR Drone 2.0 quadcopter: Investigations for improving the security of a toy. In *Mobile Devices and Multimedia: Enabling Technologies, Algorithms, and Applications 2014*, volume 9030, 2014.
[66] PX4 Autopilot Software. https://px4.io/.
[67] 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.
[68] Transparency Market Research. Robot Operating System Market - Snaphshot, 2018. https://www.transparencymarketresearch.com/robot-operating-system-market.html.
[69] 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.
[70] F. Roesner, D. Molnar, A. Moshchuk, T. Kohno, , and H. J. Wang. World-driven access control for continuous sensing. In *ACM Conference on Computer and Communications Security*, 2014.
[71] ROS.org—Powering the World's Robots. https://www.ros.org.
[72] ROS 2–ROS 2 documentation, the latest version of the robot operating system. https://index.ros.org/doc/ros2/.
[73] A. Rozantsev, V. Lepetit, and P. Fua. Flying objects detection from a single moving camera. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.
[74] The Real-time Publish-Subscribe Protocol DDS Interoperability Wire Protocol. https://www.omg.org/spec/DDSI-RTPS/2.3/PDF.
[75] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security*, 2004.
[76] Security-enhanced Linux in Android. https://source.android.com/security/selinux.
[77] S-H. Seo, J. Won, E. Bertino, Y. Kang, and D. Choi. A security framework for a drone delivery service. In *2nd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use*, 2016.
[78] D. P. Shepard, J. A. Bhatti, T. E. Humphreys, and A. A. Fansler. Evaluation of smart grid and civilian UAV vulnerability to GPS spoofing attacks. In *Radionavigation Laboratory Conference Proceedings*, 2012.
[79] H. Sun, K. Sun, Y. Wang, and J. Jing. Reliable and trustworthy memory acquisition on smartphones. *IEEE Transactions on Information Forensics and Security*, 10(12), December 2015.
[80] Nvidia Jetson TX2. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/.
[81] J. R. Vasquez, K. M. Tarplee, E. E. Case, A. M. Zelnio, and B. D. Rigling. Multisensor 3D tracking for counter small unmanned air vehicles. In *Proceedings of SPIE (conference record of the Society of Photo-Optical Instrumentation Engineers)*, 2008.
[82] E. Vattapparamban, İ. Güvenç, A. İ Yurekli, K. Akkaya, and S. Uluağaç. Drones for smart cities: Issues in cybersecurity, privacy, and public safety. In *International Wireless Communications and Mobile computing Conference*, 2016.
[83] A. Vijeev, V. Ganapathy, and C. Bhattacharyya. Regulating drones in restricted spaces. In *20th International Workshop on Mobile Computing Systems and Applications*, 2019.
[84] C. Wang, A. Carzaniga, D. Evans, and A. L. Wolf. Security issues and requirements for internet-scale publish-subscribe systems. In *35th Annual Hawaii International Conference on System Sciences*. IEEE, 2002.
[85] R. Wang, A. M. Azab, W. Enck, N. Li, P. Ning, X. Chen, W. Shen, and Y. Cheng. SPOKE: Scalable knowledge collection and attack surface analysis of access control policy for security enhanced Android. In *ACM Asia Conference on Computer and Communications Security*, 2017.
[86] R. Wang, W. Enck, D. Reeves, X. Zhang, P. Ning, D. Xu, W. Zhou, and A. Azab. EASEAndroid: Automatic policy analysis and refinement for security enhanced Android via large-scale semi-supervised learning. In *USENIX Security*, 2015.
[87] W. Wang, Y. Sun, H. Li, and Z. Han. Cross-layer attack and defense in cognitive radio networks. In *IEEE Global Communications Conference*, 2010.
[88] X. Wang, K. Sun, Y. Wang, and J. Jing. DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices. In *Network and Distributed Systems Security Symposium*, 2015.
[89] Y. Wang, H. Xia, Y. Yao, and Y. Huang. Flying eyes and hidden controllers: A qualitative study of people's privacy perceptions of civilian drones in the US. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 3, 2016.
[90] R. White, G. Caiazza, H. Christensen, and A. Cortesi. SROS1: Using and developing secure ROS1 systems. In *Robot Operating System (ROS)*. Springer, 2019.
[91] R. White, D. Christensen, I. Henrik, and D. Quigley. SROS: Securing ROS over the Wire, in the Graph, and through the Kernel. *arXiv:1611.07060*, 2016.
[92] R. White, H. Christensen, G. Caiazza, and A. Cortesi. Procedurally provisioned access control for robotic systems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2018.
[93] Y. Xu and E. Witchel. Maxoid: Transparently confining mobile applications with custom views of state. In *European Conference on Computer Systems*, 2015.
[94] Y. Yao, H. Xia, Y. Huang, and Y. Wang. Privacy mechanisms for drones: Perceptions of drone controllers and bystanders. In *ACM SIGCHI Conference on Human Factors in Computing Systems*, 2017.
[95] A. Young. Passenger jet carrying 240 people nearly hits a drone at 15,000ft, 2018. The Daily Mail, UK, 15 September 2018.
[96] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *ACM/USENIX Symposium on Operating Systems Design and Implementation*, 2006.

All URLs in references were last visited in August 2020.