Monitoring and Preventing Data Exfiltration in Android-hosted Unmanned Aircraft System Applications

Akshat Malik

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Cameron D. Patterson, Chair
Changwoo Min
Ryan K. Williams

July 24, 2019
Blacksburg, Virginia

Keywords: Android, UAS, data exfiltration

Monitoring and Preventing Data Exfiltration in Android-hosted Unmanned Aircraft System Applications

Akshat Malik

(ABSTRACT)

With the dominance of Android in the smartphone market, malware targeting Android users has increased over time. Android applications are now being used to control unmanned aircraft systems (UAS) making smartphones the storehouse for all the data that is generated by the UAS. This data can be sensitive in nature which puts the user at the risk of data exfiltration. As most Android-hosted UAS applications are proprietary software, their source code cannot be studied or modified. This thesis discusses an external monitoring system which is devised in order to assess the threat of data exfiltration. The system is further used to analyze the network behavior of the popular Android-hosted UAS application, DJI GO 4. Current methods to limit data exfiltration are discussed along with their limitations and are categorized based on the ease of deployment. Even though the Android framework provides a permission system which helps to limit the capabilities of an application, this security mechanism is coarse-grain in nature. The user either allows access to the required permissions or the application fails to function. Moreover, there is no system in place to provide a finer control over the existing permissions that are granted to an application. This thesis proposes a fine-grain and application-specific access control mechanism based on system call interposition. The solution focuses on limiting the I/O operations of the target application without any framework or application modification.

Monitoring and Preventing Data Exfiltration in Android-hosted Unmanned Aircraft System Applications

Akshat Malik

(GENERAL AUDIENCE ABSTRACT)

Advances in smartphone technology has led major consumer and commercial unmanned aircraft system (UAS) manufacturers to provide users with the feature to fly the UAS using their smartphones. The UAS generate and store large amounts of data which may be sensitive in nature. This has led the U.S. Department of Defense to ban the use of all commercial off-the-shelf UAS due to the threat of data leakage. This thesis discusses an external monitoring system which maps the network behavior of an Android-hosted UAS application, along with the existing methods to limit data leakage. To overcome the limitations of existing techniques, a fine-grain and application-specific access control mechanism is proposed. The solution provides users with the ability to enforce custom security policies to safeguard their data.

Dedication

To my family, friends and all those who have supported me in my educational endeavors.

Acknowledgments

I would like to thank my advisor, Dr. Cameron D. Patterson, for providing me an opportunity to work on this interesting project. His guidance, support and tremendous patience have been valuable to the completion of this thesis. I would also like to thank Dr. Changwoo Min and Dr. Ryan K. Williams for being a part of my advisory committee. I am thankful to my family and friends for their constant support and encouragement.

This material is based upon work supported by Naval Air Warfare Center - Aircraft Division (NAWCAD) under contract N00421-16-2-B001. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of NAWCAD.

Contents

List of Figures

List of Tables

List of Abbreviations

ARP Address resolution protocol

FTP File transfer protocol

GCS Ground control station

HAL Hardware abstraction layer

IPC Interprocess communication

MAC Media access control

MAVLink Micro Air Vehicle Link

NDK Native development kit

NIC Network interface controller

UAS Unmanned aircraft system

UAV Unmanned aerial vehicle

UDP User datagram protocol

Chapter 1

Introduction

The concept of a UAV, or drone as it is commonly known today, witnessed its inception by the Austrian Army in 1849. A UAV is an aircraft which does not have any on-board crew or passengers and can be remotely controlled or programmed to perform the desired tasks. The Austrian Army used unmanned balloons loaded with explosives in an attempt to bomb Venice. The concept was further exploited in the First World War where unmanned balloons were used for reconnaissance through aerial photography. With the emergence of winged aircraft and radio control, the first remote-controlled aircraft was created by Reginald Denny for the U.S. Army during the Second World War [16]. The UAV technology has improved significantly since then with the rise in drone warfare.

Along with their military applications, the use of commercial and recreational drones has increased in the last decade. After realizing the benefits of commercial drones, various industries worldwide have started using them for a diverse set of applications. With the rapid advance in image sensing, flight controller and battery technology, drones are now being used in climate change monitoring, search and rescue operations, photography, and will soon be employed for package delivery.

In order to target consumers and hobbyists, civilian drone manufacturers have enabled their aircraft to be controlled using a smartphone. This has allowed an average person with no prior pilot training to fly consumer drones with ease and has resulted in their widespread use. However, in doing so, the smartphone-hosted UAS application generates and stores a large

amount of data such as location or video from the drone, which can be sensitive in nature. Any instance of an unauthorized transfer or access to a user's sensitive data is termed *data exfiltration*. Since data is a valued commodity, smartphones have become vulnerable to data exfiltration just as any other computer system.

Sensitive data falling into the wrong hands can lead to consequences such as putting national security at risk [38]. With such a threat in mind, the U.S. Department of Defence has banned the use of commercial off-the-shelf drones, particularly the ones manufactured by the Chinese company DJI [41]. This apprehension can be illustrated through the example of Strava, an online fitness tracker. It is a social fitness network which allows users to share their fitness activities tracked by GPS-enabled devices. The company maintains a global activity heat map of users which can be viewed by anyone. With many Strava users working for various military and intelligence organizations around the world, security researchers were able to identify secretive military bases and patrol routes as shown in Figure 1.1.

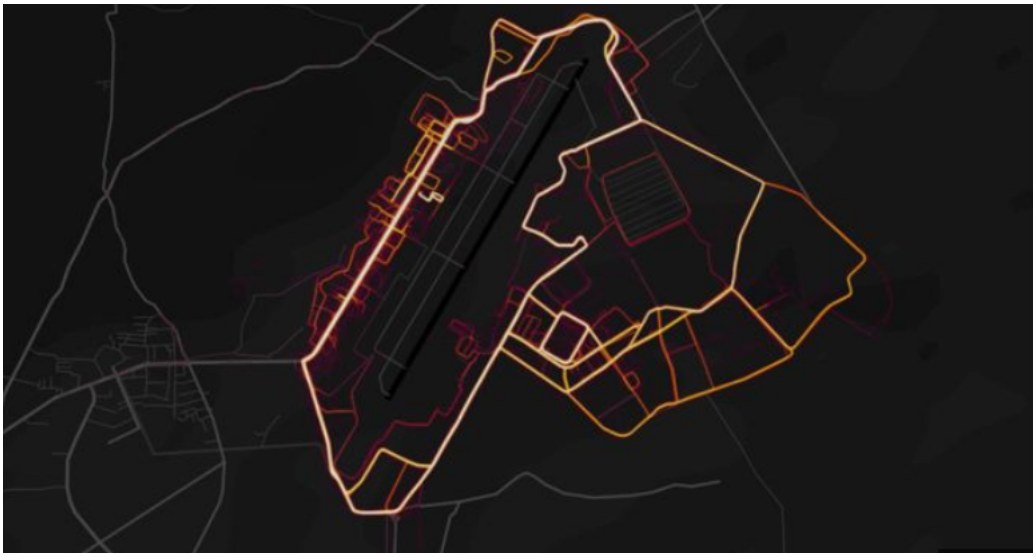

Figure 1.1: Movement of Soldiers in the U.S. Military Base in Afghanistan [21]

A recent study conducted by the National Oceanic and Atmospheric Administration on the DJI S-1000 drone revealed that the device is safe to operate. However, when the author of

this study performed the same test on his personal DJI drone, he discovered that encrypted data was being sent to unknown servers [53]. The key distinction is that the official study did not use DJI's proprietary smartphone application for controlling the drone [32].

This raises the question whether the military or an average user can trust the commercial and consumer drone manufacturers with the sensitive data generated by their equipment. Hence, it is imperative to develop ways to monitor and prevent data exfiltration in smartphone-hosted UAS applications. This thesis assesses the threat associated with using the popular smartphone-hosted UAS application DJI GO 4 and offers procedures and recommendations to limit the threat. Additionally, a standalone *tracer* program leveraging software debugging technology is proposed for enforcing fine-grain access control based on user-defined rules. As a proof of concept, the tracer program is used to monitor a simple weather application. A set of security policies are provided based on the desired behavior of the application. Any other behavior is disallowed and logged for analysis.

The target application is treated as a black box which cannot be modified as is the case with all proprietary smartphone-hosted UAS applications. As Android dominates the smartphone market, it is selected as the base operating system and DJI's popular Spark quadcopter is used as the target UAS in this research..

1.1 Contribution

The following contributions are made during the course of this research:

- An external network monitoring system to analyze the network behavior of the target Android-hosted UAS application.
- Categorization of existing techniques and procedures to limit data exfiltration based

on their ease of deployment.

- A tracer program to monitor as well as enforce a fine-grain access control security policy to prevent data exfiltration.

1.2 Thesis Organization

Chapter 2 provides background information to establish the motivation and research objectives. Chapter 3 focuses on the analysis and study of the target UAS software and hardware equipment in an attempt to find pertinent sources of data exfiltration. An external monitoring system is described in Chapter 4 that helps to analyze the network behavior of the target smartphone application, and the results from the analysis are then discussed to assess the risk associated with using the target UAS. Chapter 5 evaluates existing techniques and procedures to limit data exfiltration. Chapter 6 provides an overview of the Android architecture stack, which helps to explain the proposed solution discussed and evaluated in Chapter 7. Conclusions are presented in Chapter 9, along with future directions for this research.

Chapter 2

Background

A civilian UAV or drone is a battery powered flying vehicle without a human operator aboard. It uses aerodynamic forces to lift the vehicle and can fly autonomously or be piloted remotely. The vehicle carries a payload such as a camera or specialized sensors. The components in a UAS are:

- UAV: A pilot-less aircraft with an onboard processor, flight controller, specialized sensors, battery and actuators. Everything is put together using light but sturdy materials. Commercial off-the-shelf drones usually adopt a multi-rotor design.
- Ground control station (GCS): A ground-based control center which helps to control and manage a UAV remotely. It can either be human-operated or autonomous. The traditional radio transmitter and receiver are increasingly replaced by smartphones or tablets for ease of use.
- Communication: A radio interface is primarily used as the communication medium which forms the connection between the ground control station and the UAV. A broadband link helps to carry command, control, video feed as well as telemetry data on a single radio link. The MAVLink (Micro Air Vehicle Link) protocol [47] is commonly used in modern civilian drones for exchanging command and control data.

Various commercial and military organizations are taking advantage of inexpensive consumer and commercial drones to increase the efficiency of their tasks and to use them for safety-critical operations. Some of the important areas where drones are actively being used are:

- Inspection and monitoring: Drones reduce time, money and manpower which is spent inspecting critical infrastructure such as oil pipelines, bridges and electrical grids by using high-end camera technology.
- Search and rescue: By taking advantage of drone's aerial perspective, rescue squads around the world are able to increase their efficiency.
- Aerial photography and filming: Film makers and photographers are using drones in creative ways for their artistic endeavours.
- Condition survey and civil engineering: With their ability to reach and hover above specific sites, drones are being used for subdividing land, identifying property boundaries and surveying sites for construction.
- Climate change monitoring: Researchers are using drones equipped with high-end image sensors for environmental monitoring in areas where traditional surveying is expensive.
- Delivery: Drones are being used in the food, healthcare and postal sectors to transport food, packages and other goods.

Many drone applications involve surveying critical infrastructures or gathering intelligence for military operations which creates a cyber-espionage risk to any organization which uses them. As a consequence of this risk, the U.S. Army has prohibited its personnel from using drones manufactured by the Chinese company DJI [53]. Additionally, the U.S. Cybersecurity

and Infrastructure Security Agency has cited concerns about technology manufacturers based in an authoritarian states that permit their intelligence services to have unquestioned access to their data [22]. With a smartphone acting as the GCS, the drones have direct access to the Internet which exposes any consumer or commercial UAS to the threat of data exfiltration.

2.1 Risks Associated with a Smartphone-based GCS

In recent years, with the advances in technology, smartphones have increasingly become an integral part of our daily lives. New applications are being created every day which allow us to take pictures, share content on social media, access our bank and even help us find the quickest way to reach our destination. An average user trusts their smartphone with personal details ranging from medical information to financial credentials for a more personalized and intuitive experience. Thus, a smartphone acts as a storehouse for sensitive information and can be a target for anyone with malicious intentions. The following two case studies further illustrate this risk.

Facebook–Cambridge Analytica data scandal

In early 2018, it was revealed that Facebook's application design allowed a third-party, Cambridge Analytica, to have access to millions of users' personal information without their consent [43]. This data consisted of users' likes, current residence, birthday, etc. This personal data harvesting led to the creation of personality profiles of millions of Facebook users. Various political parties consulted Cambridge Analytica to influence public opinions by targeted political advertising, which is believed to have had an impact on the U.S. 2016 presidential election. This serves as a clear example of how personal data could be misused to have a global impact. It shows how people are seen as walking data sources by technology

giants who use their data for self-serving interests. It also raises questions about privacy and safety of a user's personal information in the hands of big technology providers.

Popular Mac application Doctor Adware

In late 2018, security researchers discovered that the popular application Doctor Adware on Apple's App store was covertly storing users' data and sending it to a server located in China [49]. The application posed as a security tool and disguised its universal access request as a malware scan permission to bypass Apple's security policy. As per the policy, an application is isolated from other applications and runs in a container called sandbox. This prohibits an application to access more resources than it needs to function unless granted permission by the user. Due to the positive reviews (likely to be fake) of Doctor Adware on the App store, a sense of trust was instilled in users. Therefore all necessary permissions were granted to the application. This allowed the application to access data from other applications and running processes. The application also maintained a log of the browsing history from all available browsers installed on the device. The security researchers have pointed out that application's suspicious activity began after an update. This shows that the developer can always add malicious hidden functions and exploit the existing permissions granted to their application.

Like Apple's permission system, Google has also equipped Android with a permission system which informs the user what resources and information the application requires. However, the caveat here is that the user has to accept all permissions otherwise the application may not be installed or function properly [9]. Therefore, even though a coarse-grain security policy already exists on Android, it is inadequate if the application exploits existing broad permissions for malicious activities. A user cannot deny the application access to selective information such as specific local files or specific Internet addresses.

2.2 Application Sandboxing

Sandboxing is a computer security mechanism to isolate a program or application by executing it in a containerized space. The mechanism creates a safe execution space for untrusted applications without jeopardizing system security. The fundamental principle behind sandboxing is to limit the privileges granted to an application in an attempt to reduce the risk.

If Android-hosted UAS applications cannot be trusted, it makes sense to execute them in a sandboxed environment. Generic sandboxing solutions available for the Android platform such as Samsung Knox [13] create an isolated environment for running applications. Even though the applications run in an isolated environment, they still have complete access to the broad permissions granted by the user.

Applications like XPrivacy prevent untrusted applications from leaking sensitive data by restricting the information that the application can access [45]. This is achieved by providing fake data to the application such as supplying an empty contact list if the application has access to the user's contacts. However, there is no ability to restrict access to just a select list of contacts. Hence, these solutions fail to provide a fine-grain access control on the permissions granted to the application and thereby cannot limit the Internet activity of the target application.

2.2.1 Fine-grain Sandboxing

System calls allow a program to interact with the operating system. These are service requests to the system's kernel from an application or process [15]. These requests require a higher privilege level to complete and therefore are managed by the kernel. All I/O operations including network requests on Android are executed using system calls. Therefore,

if an application's system calls are monitored, the application can also be monitored. The technique to monitor and modify system calls is termed *system call interposition* [39].

The tracer program proposed in this thesis uses this technique to trace all communication between the target application and the Android operating system. This not only provides a way to monitor the application but also helps to enforce a fine-grain access policy. Unlike the generic sandboxes, this acts as a *reference monitor* [24] and focuses on limiting the I/O operations of the target application. Reference monitors check the program execution for any security violations based on a defined security policy. Similarly, the tracer program monitors the system calls executed by the target application and verifies the requests based on a user-defined security policy. For example, the tracer can deny access to a specific file even though Android has granted the application permission to access the entire file system.

Chapter 3

Target UAS Analysis

Da-Jiang Innovations (DJI) is a Chinese technology company which is based in the heart of Chinese Silicon Valley, Shenzhen, Guangdong. The company is currently the world leader in the commercial and civilian UAS industry and has captured nearly 70% of the market [44]. DJI is known for their inexpensive and easy to use UAVs which employ their in-house manufactured camera gimbals, camera stabilizers, propulsion systems and flight control systems. This chapter gives an overview of the hardware architecture of the Spark drone along with the software tools provided by DJI in order to determine possible sources of data exfiltration.

3.1 DJI Spark

DJI Spark, as depicted in Figure 3.1, is part of the consumer line of DJI drones and is amongst the lowest cost drones they have to offer. It runs a Leadcore LC1860C ARMv7-A CPU with Android 4.4. Most of the components that are found in a typical Android image (as the 4.4.4 build for a Nexus 4) are not included in the DJI build. For example, the DJI software only uses core components of Android such as the runtime and debugging utilities, and also some of the camera and video processing libraries.

The device is equipped with a set of external interfaces which can be potential sources of data exfiltration:

Figure 3.1: DJI Spark

- 2.4/5.8 GHz Wi-Fi
  - Provides a medium for the DJI GO 4 application or DJI remote controller to send commands to control and fly the drone.
  - It is not used to form a direct connection to the Internet.
- USB
  - Connects DJI Assistant 2 application and the drone.
  - Can be used to access data present on the drone.
  - It is not used to form a direct connection to the Internet.
- MicroSD slot
  - Used for attaching a microSD card.
  - MicroSD card can be physically removed from the drone to access the data.

3.2 Spark Remote Controller

The Spark remote controller is used to get extra range and precision when flying the drone. The firmware is based on OpenWRT, which is a Linux distribution designed for wireless routers and embedded devices. It is equipped with a set of external interfaces which can be potential sources of data exfiltration:

- 2.4/5.8 GHz Wi-Fi
  - Used as a bridge between the DJI GO 4 smartphone application and the drone.
  - Sends and receives commands in the form of UDP network packets to control the drone.
  - Has an FTP server running mainly to transfer firmware updates to the drone.
  - It is not used to form a direct connection to the Internet.
- USB
  - Used for charging the remote controller.
  - Can be used to form a direct connection to the DJI GO 4 application to avoid the Wi-Fi interface.

3.3 DJI Assistant 2

It is an application designed to manage the DJI drone from a desktop computer and has an interface as shown in Figure 3.4. The application enables the user to connect the drone to a desktop computer via the USB interface. The user can then update the drone firmware, calibrate sensors, read flight logs and change Wi-Fi settings of the drone using the application.

Figure 3.2: DJI Spark Remote Controller

The Assistant is developed on the Electron project for building cross-platform apps. The main application executable is a compiled C/C++ application based on the Qt framework that implements a basic `WebSocket` server local to the host. The application when run launches a new process called the *browser* which is the Electron-based user interface to the application.

Communication with the Spark drone takes place using the serial device that is created on the host system. The application uses `WebSockets` to form a communication channel between the browser and assistant processes. Being a desktop application, the potential source of data exfiltration is:

- Network interface on a computer
  - Uses `WebSockets` for communication between DJI servers and assistant processes.
  - Can be used to update the drone's firmware and geofencing database.

– Can be used to share location, video or telemetry data saved on the drone with DJI or third-party servers.

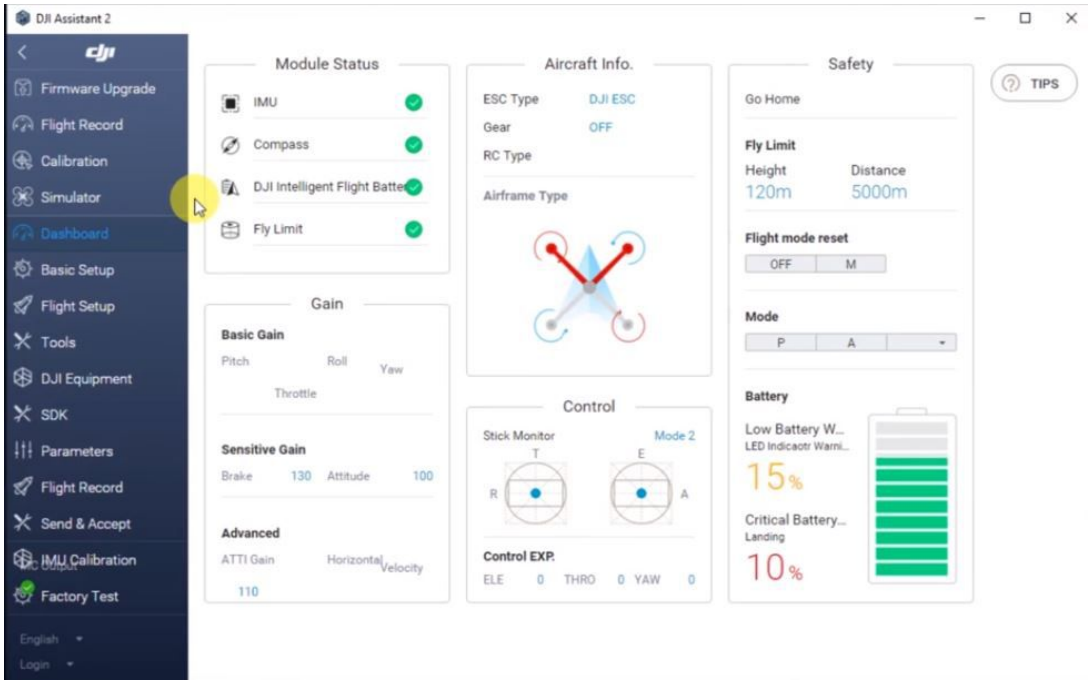

Figure 3.3: DJI Assistant 2 Interface

3.4 DJI GO 4

DJI GO 4 is an Android and iOS application for flying DJI drones and has an interface as shown in Figure 3.4. It has the feature to capture, edit, and share media content from the drones. It is used for updating the drone and remote controller's firmware. The application requires Internet access to register the drone and to receive regular updates to the geofencing database. The sensitive permissions required by the Android version of DJI GO 4 are location, phone, photos, media, camera, microphone and bluetooth/Wi-Fi.

Potential sources of data exfiltration are:

- 2.4/5.8 GHz Wi-Fi
  - Used for connecting the application to the drone for the purpose of flying.
  - Used for connecting the application to the Internet for application, firmware, and geofencing database updates.
  - Can be used to share location, video or telemetry data with DJI or third-party servers.
- Cellular network
  - Used for connecting the application to the Internet for application, firmware, and geofencing database updates.
  - Can be used to share real-time location, video or telemetry data with DJI or third-party servers while flying the drone.

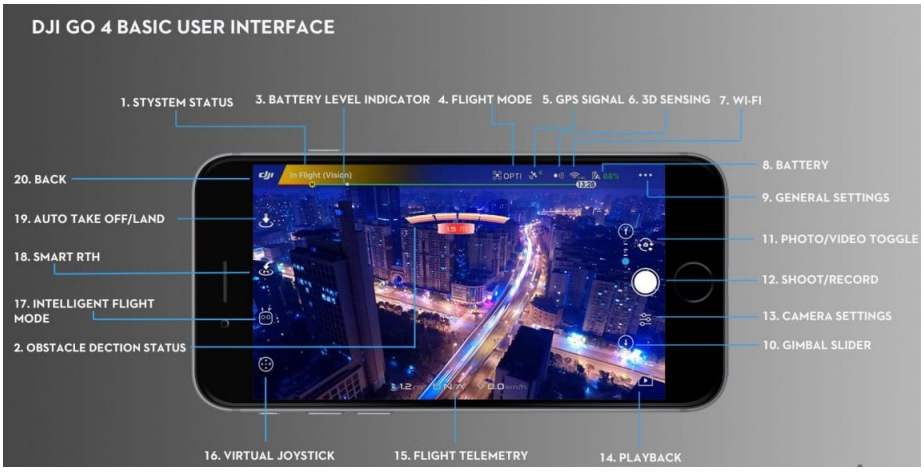

Figure 3.4: DJI GO 4 Interface

3.5 Gateway to the Internet

After the equipment analysis it can be concluded that DJI GO 4 and DJI Assistant 2 present the risk of data exfiltration as they have direct Internet access. There is the possibility of location, telemetry or video data being shared with DJI or third-party servers without the operator's knowledge or permission. The permissions granted to the application on an Android device are broad enough that automatic updates could utilize these permissions for malicious activities without any notification to the user.

Chapter 4

External Monitor

The data generated and stored by the smartphone-based GCS can be exfiltrated by an adversary who has physical access to the device or it can be transferred to a remote server by the application without the user's consent. With DJI GO 4 acting as the gateway to the Internet, it is important to monitor the network packets going to and from the application. To understand and analyze the network behaviour of DJI GO 4 without modifying the application or the Android operating system, an external monitoring system is described in this chapter.

4.1 Man-in-the-middle

A man-in-the-middle technique is normally a form of cyber attack where an adversary intercepts network communication between two parties to either eavesdrop or modify the network packets exchanged. The adversary acts as the man-in-the-middle of the communication without being detected by the targets. An external monitoring system is set up which employs this technique to capture the network communication between DJI GO 4 and the Internet router.

4.2 Implementation

A laptop or desktop computer is equipped with two network interface controllers (NICs) and acts as the sniffing device. It is configured to intercept the network traffic between the two target devices. The two NICs make it possible to reroute the network packets through the device to capture all the packets sent and received by DJI GO 4 as illustrated in Figure 4.1. A network capturing tool such as Wireshark [18] allows the device to capture, monitor and filter all network traffic going through the device. The system treats the smartphone running DJI GO 4 and the Internet router as the target devices. There are two possible ways to configure the sniffing device which are discussed below.

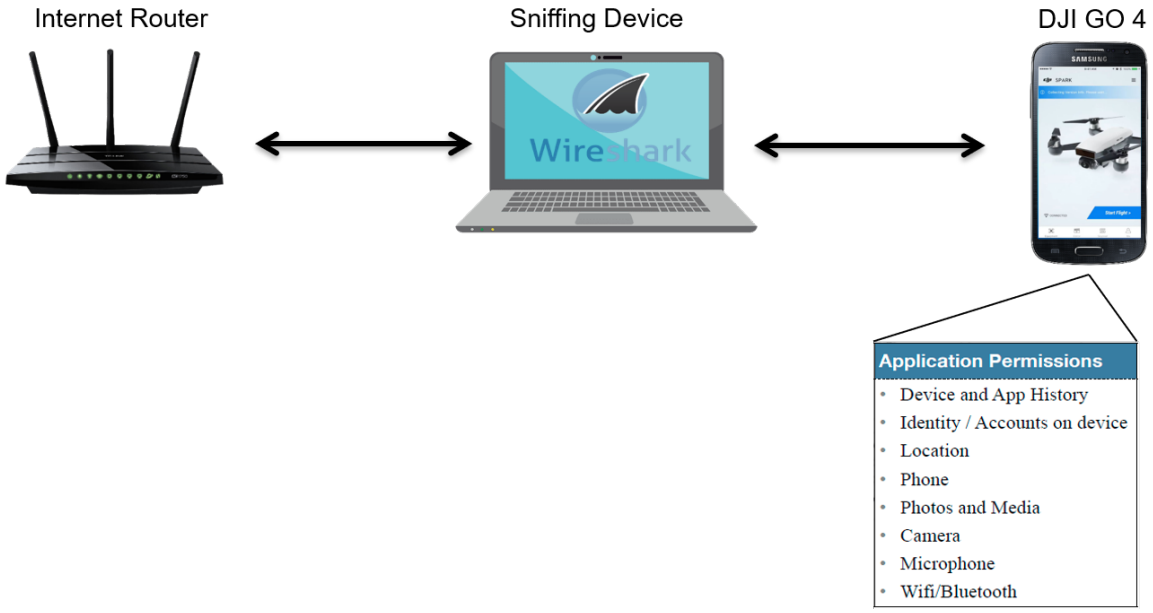

Figure 4.1: External Monitor Setup

4.2.1 ARP Poisoning

Communication on a computer network requires both an IP address and a MAC (media access control) or hardware address. ARP (address resolution protocol) is a networking

protocol used to resolve or map IP addresses to MAC addresses in a local area network. When a host on the network needs to send a packet to another host with a given IP address, it first broadcasts a request to all hosts on the network asking for the MAC address of the destination IP address. The host with the destination IP address replies with its MAC address. Important terms associated with ARP are:

- ARP request: Broadcasting a packet over the network to find the destination MAC address.
- ARP response/reply: A response packet sent by the destination host with its MAC address.
- ARP Cache: To avoid constant ARP requests, the source host saves all resolved MAC addresses in a table for future reference.
- ARP Cache Timeout: A time limit after which the ARP Cache is flushed.

ARP poisoning is a process which involves creating forged ARP response packets and flooding the ARP cache of the target host with them. Figure 4.2 illustrates an ARP packet where the operation field specifies the action requested by the sender. It could either be an ARP request or ARP reply. The sniffing device sends forged ARP packets to the Internet router as well as the smartphone. This causes the ARP cache of both target devices to change. The smartphone's updated ARP cache contains an entry linking the router's IP address with the MAC address of one of the NIC's present on the sniffing device. Similarly, the router's updated ARP cache contains an entry mapping the smartphone's IP address with the other NIC's MAC address. The device is configured with a forwarding rule to send all network packets from one NIC to the other. This enables Wireshark to capture all flowing network packets, making DJI GO 4's network behavior observable.

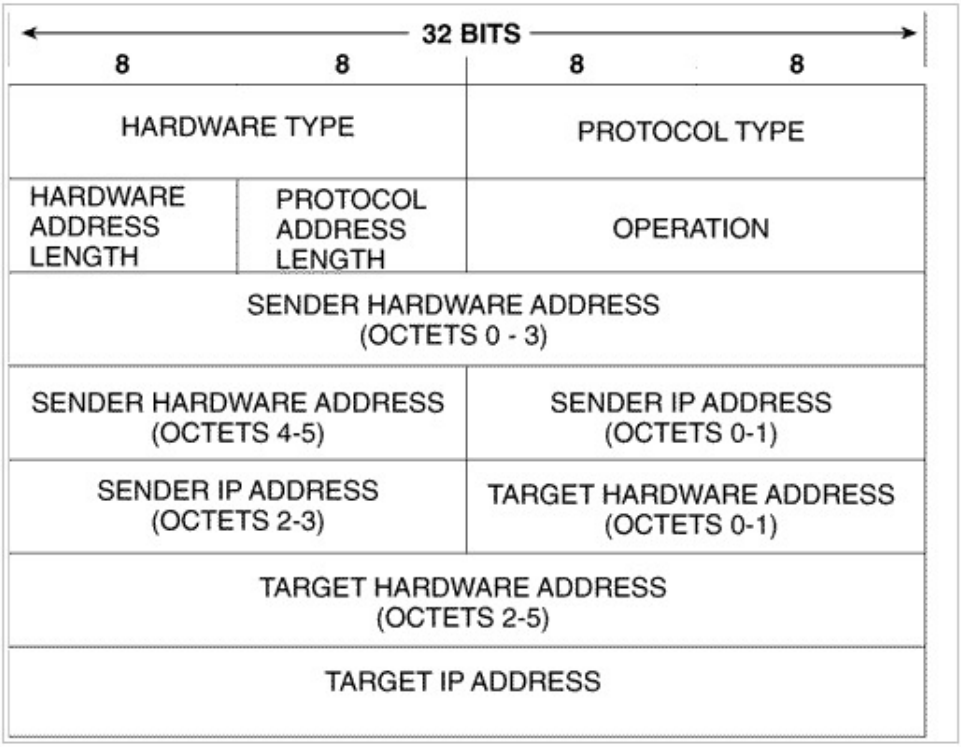

Figure 4.2: ARP Packet Format

4.2.2 Rogue Access Point

Any device equipped with a wireless network card tries to auto-connect to a stored access point that emits the strongest signal. This feature saves the user from manually connecting to a network every time the connection fails. Attackers exploit this by setting up a rogue wireless access point to lure nearby devices to join it. The victim devices easily connect to the access point as it is set up without any password. The device is configured to forward all network traffic to another network interface which is connected to the Internet. This allows attackers to covertly sniff the network packets from victim devices without disrupting their Internet connection.

The sniffing device can be configured using the same principle to act as a wireless access point. `Hostapd` is a userspace software which turns a NIC into a wireless access point. Using

this utility, one sniffing device NIC is programmed as the rogue access point, and the device is configured to forward all network traffic to the other NIC connected to the real Internet router. Wireshark is able to capture all the packets that can be analyzed to understand the network behavior of DJI GO 4.

4.3 Observations

After configuring and testing the monitoring system using the ARP poisoning and the rogue access point techniques, it is observed that ARP poisoning is not suited to long duration network monitoring tasks due to ARP cache timeout. Hence, the external monitoring system uses the rogue access point configuration to monitor DJI GO 4. It is also important to mention that the system monitors all network packets sent and received by the smartphone and cannot perform application-specific monitoring. For the purpose of this experiment, the only application installed on the smartphone is DJI GO 4 which helps prevent unwanted network traffic from any other application. The application is monitored under the following scenarios:

- Initial application login and drone authentication,
- Linking the application to the drone,
- Updating firmware and geofencing database,
- Using the application to fly the drone,
- Using various other application features,
- Application running in background but not connected to the drone.

The data is captured and stored in the form of `.pcap` (packet capture) files which are analyzed using custom filters on Wireshark. The overall analysis revealed that major network traffic is to and from Amazon Web Services (AWS) servers owned by DJI. As the evaluation is carried out in the U.S., domestic AWS servers are contacted for all authentication-related tasks. Some of the major servers communicating with the application are listed in Table 4.1. An interesting behavior is observed while the application is connected to the drone via Wi-Fi: the application continuously pings DJI servers in an attempt to connect via the cellular network. This shows that real-time data sharing with DJI servers can also take place if the user has enabled the cellular data network. Similar network analysis has been conducted by engineers managing the website `dji.retroroms.info` and the results are very similar [29].

Table 4.1: Captured Network Data

Web Address	**Location**	**Comments**
`mydjiflight.dji.com`	Virginia (AWS)	DJI service
`statistical-report.djiservice.org`	Virginia (AWS)	Tracking usage for DJI
`pro-dji-service-usa-cdn.aasky.net`	Seattle (AWS)	DJI service
`www.dji.com`	Seattle (AWS)	DJI official website
`flysafe-api.dji.com`	Virginia (AWS)	DJI GEO-related
`world.taobao.com`	Zhejiang, China	Chinese online marketplace
`stormsend.djicdn.com`	Seattle (AWS)	DJI service
`account-api.dji.com`	Seattle (AWS)	DJI account-related service
`skypixel-usa.oss-us-west-1.aliyuncs.com`	California (Alibaba)	DJI-owned video content site
`active.dji.com`	Seattle (AWS)	DJI service
`stats.jpush.cn`	Zhejiang, China	A form of push notification interface
`astat.bugly.qq.com`	Shenzhen, China	QQ is a Chinese instant messaging platform owned by Tencent
`restapi.amap.com`	Hangzhou, China	Chinese Map service

A large amount of communication with servers owned by Tencent is observed. Tencent is a Chinese multinational conglomerate that has developed the `Tinker` framework [57]. The

framework is a hot-patch system for Android. Hot-patch or hot-fix systems are dynamic repair frameworks which enable developers to push updates to their applications without releasing a new version. `Tinker` framework allows silent updates to use existing permissions in new ways not previously disclosed to the user.

According to DJI, these third-party plugins have been removed to address emerging flight security concerns as stated in a press release [5]. However, a version update to the application could revert these changes. In the same press release, DJI claims to have removed the third-party plugin `JPush` as their security researchers report that it collects unnecessary data packets including a list of applications installed on the user's device. As seen from the captured packets, connections were still being made to the `JPush` server which raises suspicion. These observations confirm the risk associated with using smartphone-hosted UAS applications and therefore warrant methods to prevent or limit any unsanctioned activity by the application.

Chapter 5

Limiting Network Access

Although the external monitoring system helps to analyze the network behavior of the smartphone, it is unable to limit unwanted network activity. This chapter provides existing techniques and procedures to help reduce the risk of data leakage by DJI GO 4. The recommendations, along with their trade-offs for the operator, are ordered according to their ease of use. Two existing advertisement blocking techniques are described, which can be customized to prevent unwanted network activity by DJI GO 4. Finally, a novel approach leveraging software debugging technology is introduced to overcome the limitation of existing techniques.

5.1 Recommendations for Drone Operators

5.1.1 Turn Off Information Sharing

In order to prevent access to device information or other related data, it is important to opt-out of DJI GO 4's data collecting programs like the product improvement program, as shown in Figure 5.1. Through this program, DJI is authorized by the user to access device information.

Limitations:

- This will not prevent DJI from covertly exfiltrating flight logs and video captures.

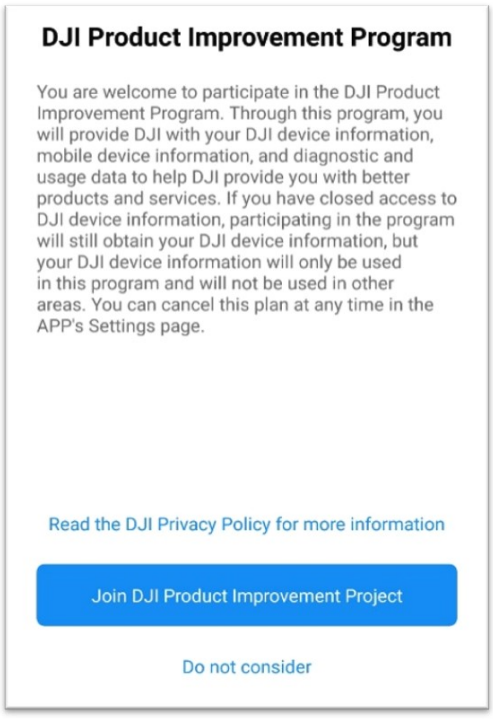

Figure 5.1: DJI Product Improvement Program Description Page

5.1.2 Disallowing Data Transfers from the Drone to DJI GO 4

Downloading media present on the drone to the application makes it more susceptible to be exfiltrated as the application has direct access to the Internet.

Limitations:

- The media content would need to be accessed by physically removing the memory card present on the drone.
- This technique is not secure against exfiltration of flight logs and cached data stored by the DJI GO 4 application on the mobile device.

5.1.3 Airplane Mode

Operators using the application can put their mobile devices in airplane mode to suspend Internet connectivity. This will prevent the application from using the cellular network to share data in real-time with DJI or third-party servers during flight mode. The application will also not be able to detect the operator's location.

Limitations:

- The real-time map which shows up in flight mode will not work.
- Some of the safety features provided by DJI will not work.
- This technique may fail as soon as the airplane mode is turned off when the application gets access to the Internet, allowing cached information collected during the flight to be exfiltrated.
- The operator will not be able to update the drone's firmware, geofencing database, or the application.
- Bugs present in the application or the drone firmware will persist.

5.1.4 Airplane Mode (OTG Hack)

The DJI Spark does not officially support this feature, but by using an older version of DJI GO 4, the USB OTG mini cable can be used to connect the mobile device with the remote controller to avoid using Wi-Fi as the communication medium. This results in higher bandwidth communication between the application and the drone, and allows the operator to switch off all network interfaces on the mobile device.

Limitations:

- By turning off all network interfaces, this technique suffers from the same trade-offs as mentioned for the airplane mode.

- USB OTG mini cable is not provided by DJI.

5.1.5 DJI's Local Data Mode

After hearing concerns related to data leakage, DJI released a feature for government and enterprise customers called the Local Data Mode. The mode claims to disallow the application from using the Internet in order to enhance data privacy assurances. This mode allows the operator to have all network interfaces active and available to other applications on the mobile device. However, this mode is unavailable on current versions of the application.

Limitations:

- By suspending the application's Internet connectivity„ this technique suffers from the same trade-offs as mentioned for airplane mode.

- This technique may fail as soon as the local data mode is turned off when the application gets access to the Internet, allowing cached information collected during the flight to be shared with DJI or third-party servers.

5.2 Advertisement Blocking Mechanisms

In the age of online advertising, some people use advertisement blockers to avoid pop-ups and banner advertisements that disrupt Internet browsing. The aim of an advertisement blocker is to restrict unwanted network connections. The majority of blockers work by maintaining a

blacklist of advertisement serving links which causes the corresponding network connections to be blocked. The same strategy can be applied to limit DJI GO 4's network activity.

5.2.1 Host File Loopback

Internet users browse the web using domain names such as `google.com` or `wired.com` whereas a computer system uses unique IP addresses assigned to each of these websites in order to access them. The domain name system (DNS) can be compared to the yellow pages of the Internet. It is used to look up a website's unique IP address by using its domain name and thereby eliminates the need for users to know a website's IP address.

When a network request is initiated, the first step taken by the system is to query a DNS server for domain name resolution. The DNS server resolves the query and responds by sending the IP address associated with the requested domain name. To save time, these mappings are cached in the `host` file present in the system directory of an Android smartphone. Before making another DNS request, Android first looks up the `host` file present locally. Remote DNS servers are queried only if the local mapping is not found. The process in action can be seen in Figure 5.2.

The `host` file loopback mechanism works by modifying this `host` file to replace the mapping of blacklisted domain names with local loopback IP (`127.0.0.0`). Whenever a network request is generated for any of the malicious domains, the local loopback IP is provided which inherently blocks the network packet from heading to its destination. By analyzing the DJO GO 4 network traffic using the external monitoring system described in chapter 4, a blacklist of suspicious domain names can be created. This can then be used to modify the `host` file to restrict DJI GO 4's network activity.

Limitations:

- Requires root access to modify the `host` file,
- If the malicious server's IP address is hardcoded, no DNS request would be generated,
- Does not provide network monitoring feature.

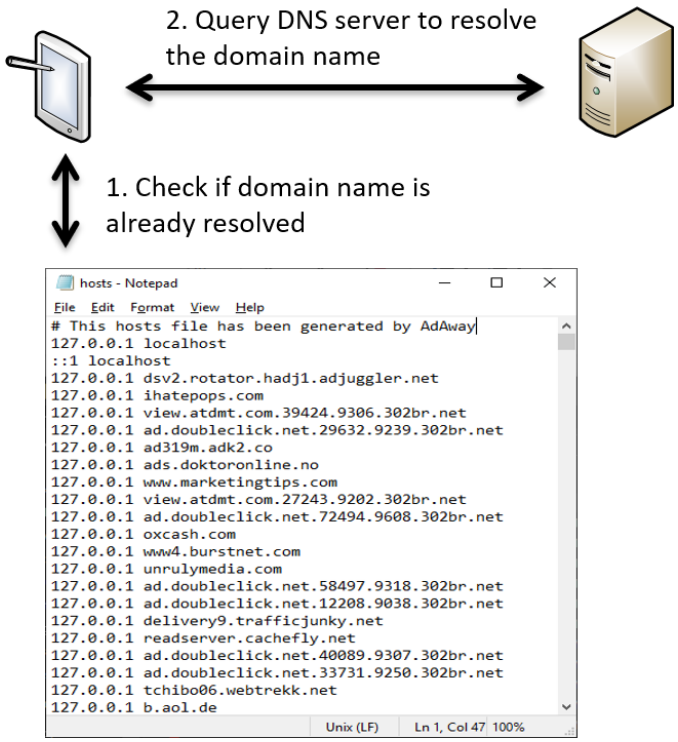

Figure 5.2: Domain Name Resolution on Android

5.2.2 VPN Gateway

A virtual private network or VPN is a widely available service which routes all network traffic generated by a device through a privately owned server to protect users' privacy. The network data is encrypted by the service to avoid any snooping attempts by malicious agents. Major VPN service providers have a built-in advertisement blocking mechanism. All the network packets routed through the private server are compared with a blacklist containing ad-serving domains to block all advertisement requests.

VPN service providers develop their VPN client for Android using existing APIs, which allows their application to act as a VPN gateway. When the client application is active, all the network traffic generated on the smartphone is routed through the gateway, as illustrated in Figure 5.3. System-wide advertisement blocking applications on Android use this feature to block pop-up and banner advertisements on all applications installed on the device [28].

Similarly, this feature can be used to monitor and restrict the network activity of DJI GO 4. By using the VPN service APIs on Android, all network traffic to and from DJI GO 4 can be inspected and compared against a blacklist to block connections to suspicious domains. Unlike `host` file loopback method, this method serves the dual purpose of monitoring as well as blocking the network activity of DJI GO 4.

Limitations:

- Unable to perform application-specific network monitoring and filtering.

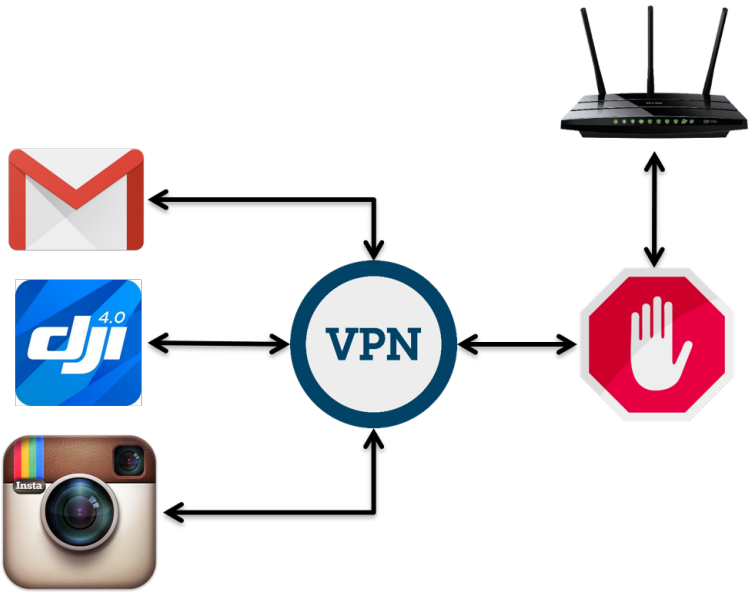

Figure 5.3: VPN Restricting Network Activity on Android

5.3 Software Debugging Mechanism

A debugger is a software tool which assists a developer to examine a program's behavior. Debuggers are used to track down any defects in a program by observing its runtime behavior. At any desired point, the debugger can halt the program's execution and inspect its state to verify its correctness. However, as programs are compiled into machine code, inspecting each line of code using a debugger is not a trivial task for a developer. Therefore, to make it easier to debug complex software programs, debuggers are able to map the machine code back to the original program source code. This is achieved by embedding comprehensive information about the source and its relation to the machine code. This technique is termed *source-level symbolic debugging* [54] and is used frequently for analyzing user applications.

```
(gdb) break main
Breakpoint 1 at 0x8048426: file hello10.c, line 6.
(gdb) run
Starting program: /home/gary/hello10

Breakpoint 1, main () at hello10.c:6
6               for(i=0;i<10;i++)
(gdb)
```

Figure 5.4: Using `gdb` to Debug a Program

Debuggers that use this technique rely on a symbol table which connects the machine code instructions to their corresponding variable or line in the source code. A debug flag must be set during the compilation process in order to build the symbol table. This table is used to step through a program line by line and set breakpoints to pause the program execution as shown in Figure 5.4. A breakpoint is a special instruction inserted at specific locations in a program's code which raises a software interrupt. In the event of such an interrupt, the CPU stops the execution of the program and hands over the control to the debugger which can

then examine or modify the state of the stopped program. This makes debuggers a useful tool to monitor and modify a program's runtime behavior.

5.3.1 Runtime Monitor Using System Call Interposition

All software applications used by consumers are compiled without the debug flag. The lack of a symbol table and extra debug information makes it difficult to observe their runtime behavior using a debugger. System call interposition is a technique to monitor and modify system calls, as described in Section 2.2.1. An application's I/O operations can be monitored by tracing the system calls issued by it. Since under the hood common debuggers like `gdb` [6] use the system utility `ptrace` [12] to create breakpoints and observe the program state, this utility can be used to halt an application whenever a system call is executed. Thus, this creates an opportunity to monitor or modify the application's runtime state.

Some of the major system calls which facilitate network-related tasks on Android are:

- `SOCKET`: Creates an endpoint for communication,
- `ACCEPT`: Accepts a connection on a socket,
- `CONNECT`: Initiates a connection on a socket,
- `LISTEN`: Listens for connections on a socket,
- `RECVFROM`: Receives a message from a socket,
- `SENDTO`: Sends a message on a socket.

By using the `ptrace` utility, these system calls can be intercepted and examined. This makes it possible to monitor the network packets sent or received by the application. The arguments

of these system calls can be modified to block blacklisted network requests and thereby limit the network activity of the target application. If the developers push an update for the application, policies can be modified to maintain security based on the changed behavior. This technique is used to develop the tracer program described and evaluated in Chapter 7.

Chapter 6

Android Operating System

This chapter provides an overview of Android's architecture stack and security policies before discussing the implementation of the tracer program presented in Chapter 7.

6.1 Architecture Overview

Android is an open source operating system whose foundation is based on the Linux kernel. It is structured as a software stack where the kernel forms the bottom layer as shown in Figure 6.1. The kernel provides the device drivers required for the vast majority of applications and features provided by Android smartphones. When the user wants to do anything requiring the use of hardware, a request is sent to the kernel. From making a phone call to increasing screen brightness of the device, anything hardware-related is controlled by the kernel. These requests to the kernel are made in the form of system calls.

Android uses a customized version of the Linux kernel which has some additional features important for a mobile embedded platforms such as interprocess communication (IPC) binder drivers. The binder framework provides a remote procedure call mechanism which allows client processes (applications) to send or receive information and execute methods in server processes (system services) such as Wi-Fi and telephone.

The Hardware Abstraction Layer (HAL) is present above the kernel layer and forms a bridge

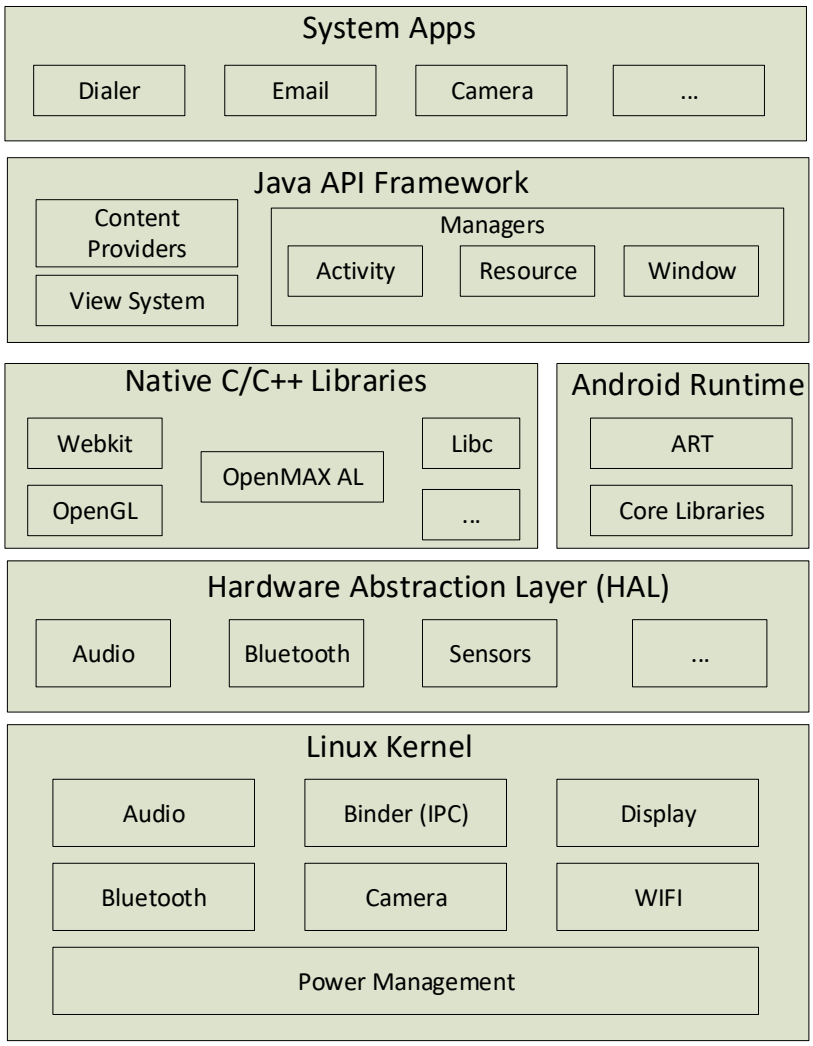

Figure 6.1: Android Stack

between the hardware and the software. The Android framework and applications sitting at the top of the stack communicate with the hardware using the Java API framework. Since, the kernel takes requests in the form of system calls, the HAL provides a standard interface allowing the Android framework to communicate with the hardware resources provided by the smartphone manufacturer [1].

Android Runtime (ART) is an application runtime environment/virtual machine which translates the application bytecode into native instructions which are then executed by the

device's runtime environment. The Android platform also provides native libraries written in C and C++ to support system components built using native code. These libraries can also be accessed by applications using the Android Native Development platform. Just below the application layer is the Java API Framework which provides services to applications in the form of APIs written in the Java language. These APIs enable access to all the services and components provided by the operating system [10].

6.2 Kernel and System Security

The Android platform offers the security features provided by the Linux kernel. Being a multi-user system, Linux follows a user-based permission model wherein one user cannot modify or access another user's files. In Android, each application is treated as a user and assigned its own user id (UID). This ensures application isolation to protect the system from malicious applications and is a form of kernel-level application sandboxing. Android versions 4.3+ use *SELinux* to describe and enforce boundaries on an application [14].

File system permissions ensure that an application's private files cannot be accessed by other applications unless explicitly shared by the developer. Android's kernel, system libraries, application runtime and framework are all contained on the system partition, which is set to read-only. IPC mechanisms allow secure communication between different applications [8]. Additionally, new Android versions (8.0+) use a `seccomp` [20] filter to block specific system calls which can be exploited by malware. However, this does not prevent an application from using the whitelisted system calls such as `sendto` and `ioctl` for malicious purposes.

6.3 Application Security

Android applications are mostly written in Java which is compiled to Dalvik byte code and assembled as `.dex` files. The bytecode is then translated to native machine code and executed by either the ART or the Dalvik virtual machine. All `.dex` files are packaged together along with configuration files as an APK (Android Application Package).

The Android permission model allows an application to only access a limited range of system resources. APIs providing access to sensitive functionality must be specified by an application in a `manifest` configuration file. During the installation phase as well as during an application's runtime (since Android 6.0), these permissions must be approved by the user. A security exception is thrown whenever an application tries to access protected APIs which are not declared in its `manifest` [3].

These protected APIs have broad functionality. For example, an application trying to access the network or data connection API will either get access to the complete Internet or not. There is no mechanism to deny permission to specific Internet addresses. For the file system API, permission cannot be denied to specific files. Hence, the Android permission model is coarse-grain in nature.

6.4 Android Boot Process

The Android boot process has the following steps:

1. Every Android smartphone starts by pressing the power button, which initiates the `bootrom` code. This is the first code executed and is present in a write-protected flash memory embedded inside the processor. The `bootrom` is responsible for loading the

`bootloader` into the RAM and executing it.

2. The `bootloader` is a program which is not part of the Android OS and is responsible for the manufacturer's specific locks and restrictions. The `bootloader` starts by detecting the external RAM and moves on to set up low-level memory management, network and security options. The `bootloader` is responsible for loading all OS-specific files.

3. Similar to Linux, the Android kernel executes by setting up the cache, protected memory, and loads all the drivers. It starts the first process of the system termed `init`.

4. The `init` process is responsible for mounting directories such as `/sys, /dev` and `/proc`. It runs the `init.rc` script and starts native daemons such as `bootanimation`, `Zygote`, `Service Manager` etc. At this stage, the Android logo animation pops on the screen.

5. `Zygote` is a special Android process which is part of the ART environment. It is responsible for starting new applications by forking itself and loads all classes and runtime required by an application into system memory to minimize startup time.

6. The first process started by the `Zygote` is the `System Server`. It initializes every system service on Android and registers them with `Service Manager` started by the `init` process.

7. The `Activity Manager` is started by the `System Server`. It creates a new Activity thread process, and maintains the Activity lifecycle and Activity stack. Finally, it receives `onClick` events to start new applications.

6.5 Application Installation and Runtime

An APK is required to install an application. This APK can either be downloaded directly from Google's application market or a third-party source. When a user initiates the APK install process, the `Package Manager` parses the APK file to display the permission requests written in the application's `manifest` file. After the user grants the permissions, the package is queued with other packages waiting to be installed. The newly installed application is provided with a unique UID and a private directory for file storage. The `Package Manager` service stores basic information regarding all the installed applications and can be queried for this information [50].

An application is launched when the user clicks its corresponding icon. This click event results in a call to the `Activity Manager` service via the Binder IPC. Information regarding the application is obtained by querying the `Package Manager`. If the application is already running, the existing process is resumed otherwise a new process is created. The `Zygote` process uses the `fork` operation for creating new processes. This operation results in a new process id (PID) to be assigned and bound to the launched application. The process then loads the application-specific code and brings the application activity to the foreground [40].

Chapter 7

Tracer

The fundamental idea behind developing the tracer program is to monitor and control the behavior of the target application. As all hardware-related tasks on Android are handled by the kernel, the target application uses system calls to communicate with it. The tracer employs system call interposition to observe the system calls made by the target application. If undesired behavior is observed, the program prevents the request and logs the attempt, thus making it easier for users to keep a constant check on the application. This chapter discusses the tools used to develop the tracer program and looks at its implementation and evaluation.

7.1 Platform and Tools

7.1.1 Android Native Development Kit

Android applications are generally written in high-level programming language such as Java or Kotlin to exploit their object-oriented design. As mentioned in Section 6.3, Java code is executed in a virtual machine and hence is platform architecture agnostic. However, to interact with the kernel, architecture dependent programs are required. Android's Native Development Kit (NDK) provides tools to develop and manage programs at Android's native layer using the C or C++ language.

The tracer program is written in C and uses the platform library support provided by NDK. The `ndk-build` script is used to compile the tracer program. The script generates native binaries using the information provided in the build configuration files `Android.mk` and `Application.mk`.

7.1.2 Android Debug Bridge

Android Debug Bridge (ADB) is a command-line utility present in the Android Software Development Kit (SDK) package. It allows the developer to form a client-server connection with the Android device, and has three main components:

- Client: Sends commands to the device and is invoked by the `adb` command on a terminal.
- `adbd`: A daemon process on the Android device which executes the commands received from the client.
- Server: A background process on the development machine that manages the communication channel.

ADB is used to gain access to the Unix shell on the Android device to transfer and execute the tracer program.

7.2 Implementation

The foundation of Android is based on the Linux kernel, therefore a utility called `ptrace` is available to monitor and debug native layer processes. `ptrace` is available in the form of a

system call which can be used by a process to inspect and manipulate the internal state of another process at runtime. It is the primary mechanism which debuggers employ to monitor target processes on Unix-like systems as discussed in Section 5.3.

The tracer program uses this to dynamically trace any system call issued by the target application. The `ptrace` API is

```
long ptrace(int request, pid_t pid, void *addr, void *data);
```

where `pid` is the target application's process ID. The request field selects a specific `ptrace` function, some of which are listed in Table 7.1. The value for `addr` and `data` depends on the type of request being passed. `PTRACE_SETOPTIONS` allows to set specific execution requests for `ptrace`. The options required for the tracer program are listed in Table 7.2.

Table 7.1: Ptrace Request Fields

Request	**Function**
`PTRACE_ATTACH`	Attach to the process specified in `pid`
`PTRACE_SYSCALL`	Stop at the next system call entrance or exit of the target process
`PTRACE_GETREGS`	Get a copy of the target process's general-purpose registers
`PTRACE_SETOPTIONS`	Set a number of options for tracing the target process
`PTRACE_PEEKUSER`	Read a word at offset `addr` in the target process's `USER` area
`PTRACE_POKETEXT`	Write a word to the address `addr` in the target process's memory
`PTRACE_SETREGS`	Modify the target process's general-purpose registers
`PTRACE_DETACH`	Detach from the target process

Table 7.2: Ptrace Setoption Fields

Option	**Function**
`PTRACE_O_TRACESYSGOOD`	Stops target process at system call occurrences by setting bit 7 of signal trap (`SIGTRAP | 0x80`)
`PTRACE_O_TRACEFORK`	Stops target process before it executes a `fork()` system call
`PTRACE_O_TRACECLONE`	Stops target process before it executes a `clone()` system call

The tracer works by using the `PTRACE_ATTACH` request to attach to the target application, using its `pid`, and temporarily becomes the parent process of the application as shown

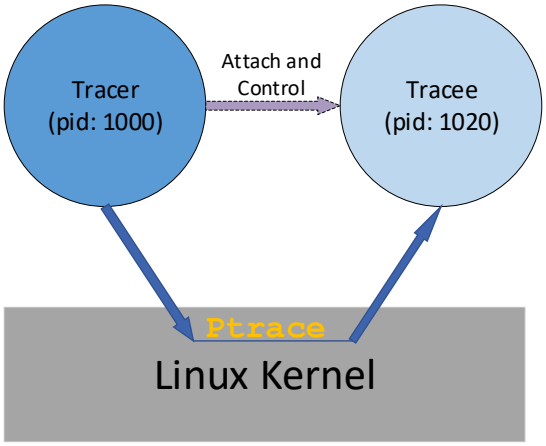

Figure 7.1: Tracer Attach Operation

in Figure 7.1. `ptrace` permits the tracer to attach to multiple process threads but each individual thread can have just one process tracing it.

When the target application is traced using `ptrace`, the tracer is able to suspend the application's execution depending on the option specified by `PTRACE_SETOPTIONS`. In the event of a system call execution, the target application receives a `SIGTRAP` signal which halts its execution before the system call is serviced as shown in Figure 7.2. Since the tracer is the parent process, it monitors this using the standard `waitpid` system call available on Android. The tracer is then able to inspect the general-purpose registers of the application process using `PTRACE_GETREGS`.

Depending on the architecture, the arguments for the system call are stored in the target process's general-purpose registers. Table 7.3 lists the registers used to store the arguments for system calls executed on ARM 32-bit and 64-bit architecture.

Table 7.3: System Call Argument Registers

Architecture	**System call number**	**Return value**	**Argument 1-6**
ARM32	r7	r0	r0-5
ARM64	x7	x0	x0-5

The system call arguments can be modified by changing the values stored in the registers. PTRACE_SETREGS allows the tracer program to modify the values of the registers thereby altering the application behavior. This allows the tracer to monitor every system call and block unwanted behavior based on user-defined rules.

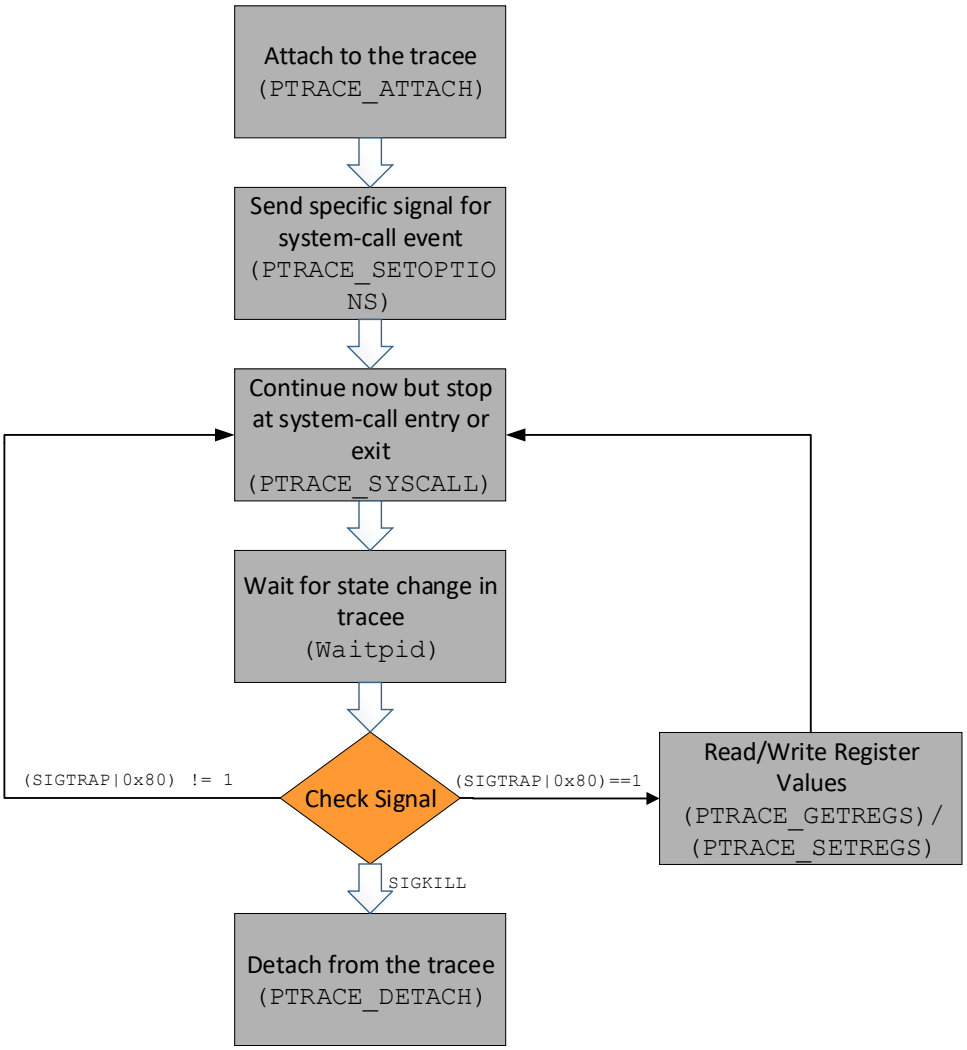

Figure 7.2: ptrace Flow Diagram

PTRACE_O_TRACEFORK and PTRACE_O_TRACECLONE options need to be set to enable the tracer to trace all new threads spawned by the target application at runtime. It is important that every thread is traced because the majority of Android applications hand over their network

operations to a spawned thread as an `AsyncTask`. Additional threads would have been created to perform different tasks if the target application is already running before being traced. The tracer manually attaches itself to every thread by iterating over the task IDs (TID) using the `proc` file system [11] to trace these threads.

7.3 Evaluation

The correctness of the tracer is evaluated by testing it on a simple weather application which performs the following steps:

1. Get a zip code from the user.

2. Fetch the weather information for the zip code using weather API calls.

3. Display the weather information received from the weather server.

The tracer and the target application are executed on a Moto G smartphone running Android 7.1.2 on the Arm v7 32-bit instruction set. A custom security policy is provided to the tracer in the form of blacklisted zip codes. The arguments of the `sendto` system call are monitored using `PTRACE_GETREGS` as shown in Figure 7.3. Zip codes being queried are compared with the ones present in the blacklist. If a blacklisted zip code is encountered, `sendto`'s message argument is modified using `PTRACE_SETREGS` which results in the query's zip code to be replaced with a default safe zip code. In Figure 7.4, the weather application on the left displays the temperature for the blacklisted zip code when the tracer is inactive. The right side displays the temperature for the default zip code when the tracer is active. This verifies that the network request can not only be monitored but also modified by the tracer.

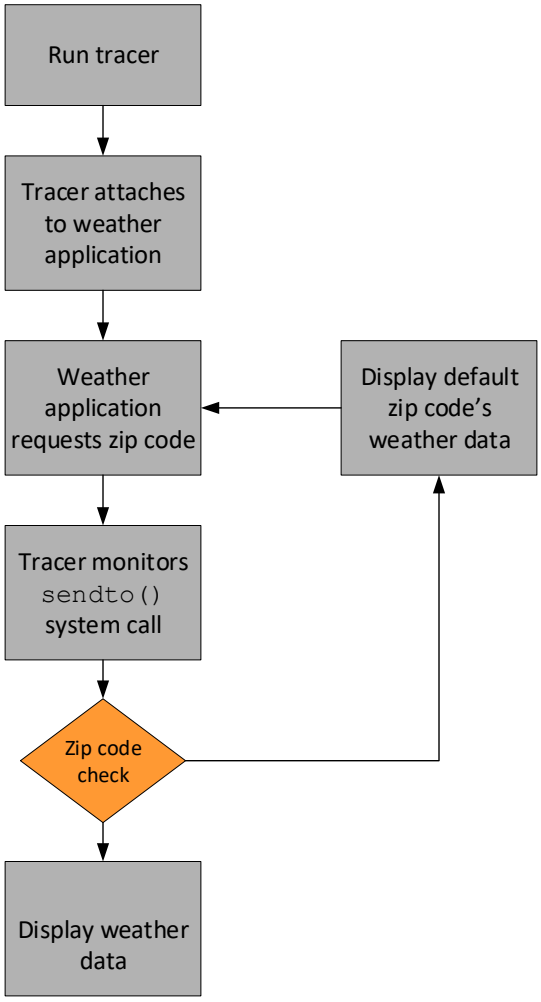

Figure 7.3: Monitoring a Weather Application

7.3.1 Performance

The performance impact of monitoring the simple weather application using the tracer is measured by referencing the `proc/stat` file. This file maintains information about the kernel activity since the system first booted such as the amount of time the CPU spends performing different tasks. This information is used to calculate the CPU utilization. The file also provides the number of context switches performed and the time spent waiting for I/O operations by the CPU.

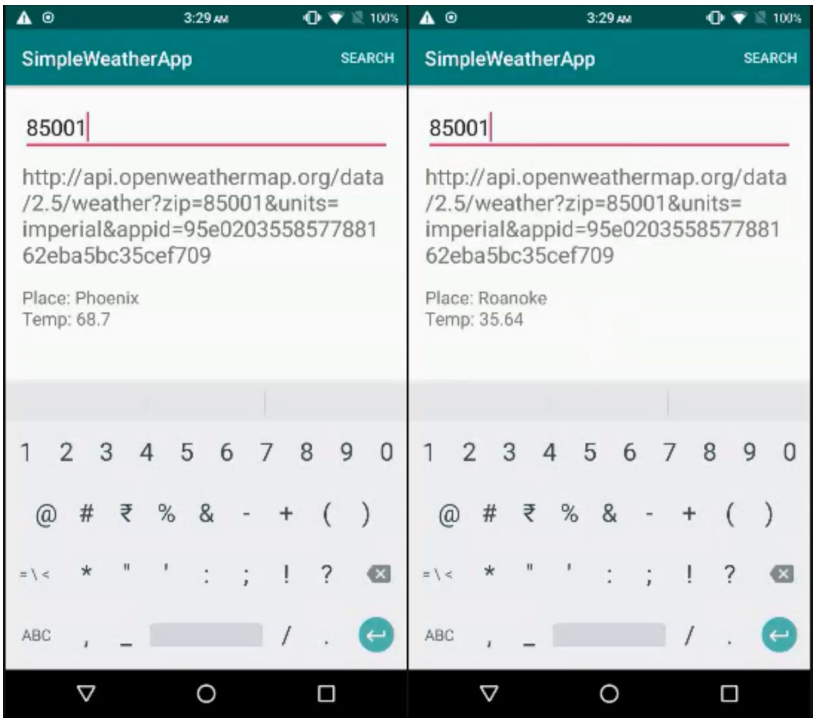

Figure 7.4: Tracer Operation

Table 7.4 shows the performance overhead of using the tracer program to enforce user-defined security policy on the weather application. The time consumed by a network request when the tracer is active is compared to when it is inactive. It is inferred that despite having a low CPU overhead, the system spends more time waiting for I/O operations when the tracer is operational. This is because the time spent waiting for I/O operations is proportional to the rate at which system calls are made which is further linked to the high amount of context switches due to the `ptrace`-based system call interposition. The time taken to execute a network request is significantly more due to the extra work spent to inspect and modify the network-related system calls made by the weather application. The results show an added overhead of using the tracer program but no performance impact was observed while using the application.

Table 7.4: Performance Overhead

Factor	Tracer Off	Tracer On	Overhead
CPU Utilization	28.085%	28.177%	0.327%
I/O Wait	85.2 jiffies[1]	101.4 jiffies	19.01%
Context Switches	77,567	236,517	67.2%
Time Taken for a Network Request	92.3ms	150.5ms	38.67%

7.4 Using the Tracer with DJI GO 4

When the tracer program tries to attach to the DJI GO 4 process using `PTRACE_ATTACH`, it gets an `operation not permitted` error. The developers at DJI have used anti-debugging techniques to disallow debuggers or other programs using `ptrace` from observing the application's runtime behavior. These techniques are presumably used to prevent any reverse engineering efforts.

7.4.1 Anti-debugging Mechanisms

Anti-debugging techniques are used by programs to ensure that they are not being inspected by a debugger. An Android application is debugged either on the Java-level using JDWP (Java Debug Wire Protocol) or on the native layer using `ptrace`. Given that the tracer operates on the native layer, the anti-debugging techniques discussed here are focused on countering native layer debugging. Anti-debugging techniques can either be preventive or reactive in nature. Preventive techniques disallow a debugger from attaching to an application while reactive techniques trigger a set response by an application. To build a strong defense, a mixture of both these techniques are used.

Self-debugging

Self-debugging is a `ptrace` feature which is used as a preventive anti-debugging mechanism by DJI GO 4. This technique exploits the fact that at a given time, only one process can attach to another using `PTRACE_ATTACH`. As mentioned earlier, a process thread can only be traced by one process. The operation is disallowed if another process tries to attach. DJI GO 4 exploits this technique to avoid programs with debugging capabilities to attach to its main process thread. As soon as the application starts, it clones itself to create a dummy child process whose task is to trace the main process thread as illustrated in Figure 7.5.

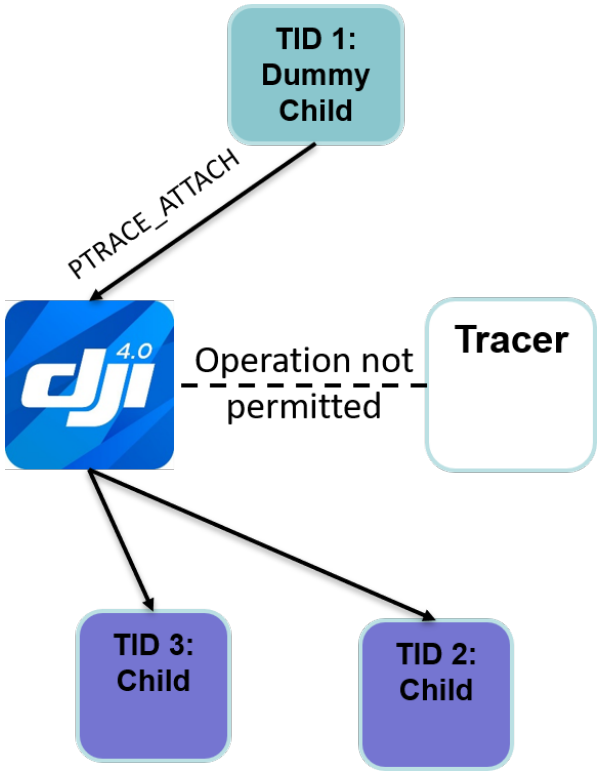

Figure 7.5: DJI Using Self-debugging

Monitoring Thread Status

Each process thread's status can be read using the `proc` file system. `TracerPID` is one of the fields present in the file `/proc/pid/status` which keeps track of the PID of the process tracing the thread. DJI GO 4's main process thread continuously monitors this field for all its child threads to ensure that they are not being traced by another process. As a reactive measure, a `kill` signal is issued to terminate the application if tracing is detected.

7.4.2 Countering Anti-debugging Mechanisms

A simplistic way to counter self-debugging would be to kill the child process tracing DJI GO 4's main process thread. This would allow the tracer program to attach to the main process. However, the child process is monitored by the main process and killing the child process causes the main process to kill itself.

As mentioned in Section 6.4, `Zygote` is the parent process of every application running on an Android device. Therefore, the tracer is able to attach to all new children processes spawned by attaching to the `Zygote` process. As a result, the tracer is able to attach to DJI GO 4's main process thread when it starts. This makes it possible to trace every system call executed by the application before the detection mechanism kicks in.

After attaching to the main process thread, the next step is to intercept all `kill` system calls and modify the arguments to prevent the application from exiting. This results in an unknown control-flow state and the application gets stuck in a boot loop. The alternate approach is based on evading the detection mechanism. In other words, to avoid generating any `kill` signals, the tracer needs to identify and modify certain system calls issued by the application such that the tracer's presence is not detected.

The following two major observations are drawn from analyzing the system calls issued by the application before it detects the tracer program:

1. The child as well as the main process keep monitoring the `/proc/pid/status` file where `pid` is the PID of the main process. If an unknown value of `TracerPID` is detected then a `kill` signal is sent to the main process.

2. In order to attach to the main process thread, the child process uses `ptrace`. However, the `ptrace` request fails as the tracer is already attached. This again results in a `kill` signal to be generated.

The tracer handles the first event by intercepting the `read` system call. By modifying the value of `TracerPID` in the read buffer, the application is tricked into believing that the tracer is not attached. The second event also requires the use of system call interposition to modify the return argument of the `ptrace` system call. By returning a success signal, the child process believes that it has successfully attached to the main process.

However, despite countering the observed detection mechanism, the application is unable to bypass the initial load phase. Therefore further analysis is required to completely counter the anti-debugging mechanism used by DJI GO 4. Section 9.1 discusses additional techniques and methods to proceed forward. By countering the anti-debugging mechanism, the tracer program will be able to trace all the system calls issued by DJI GO 4. Similar to tracing the weather application, the arguments of system calls such as `sendto` and `connect` could be monitored and modified to enforce a user-defined security policy on DJI GO 4 to limit its networking activity.

7.5 Limitations

The major limitation arises from `ptrace`'s attachment rules. The tracer must have a higher privilege level in order to attach to the target application. Hence, the tracer must have root privileges for the attach operation to be successful. Additionally, anti-debugging mechanisms require reverse engineering to understand and identify the traps set by the target application. The initial application behavior helps to understand the detection mechanism which can be countered using system call interposition. However, extensive reverse engineering effort and complex patchwork are required if complex traps are set as discussed in Section 7.4.2.

The recurring cost associated with the tracer is the modification required to the user-defined security policy if the target application receives an update. There is also an added overhead to employ the tracer for enforcing a fine-grain access policy, as discussed in Section 7.3.1. However, it is important to note that stricter security comes at a cost.

Chapter 8

Related Work

In this chapter an overview of relevant work aimed at improving Android application security is provided. Several testing and analysis techniques which help in detecting unsafe applications are discussed followed by existing research on Android application sandboxing.

8.1 Static Analysis

Static analysis is an automated analysis method to find vulnerabilities in the program source code before the program is executed. Static analysis frameworks and tools for Android [4, 31, 46] provide a way to map the expected behavior of an Android application. These tools analyze the application bytecode to detect any malicious behavior. AndroidLeaks is a static analysis framework developed by Gibler et al. [35] which creates a call graph of an application's bytecode and then performs reachability analysis to determine if sensitive information may be sent over the network. It further uses dataflow analysis to determine if the data reaches a network sink or not. Similarly another framework developed by Klieber et al. [42] detects malicious information flow on Android when different applications interact with each other. It analyzes both dataflow within as well as between different applications. However these frameworks may fail to find vulnerabilities introduced at runtime.

Some applications may try to dynamically load malicious code during runtime to avoid detection by static analysis. Although, the static analysis tool proposed by Poeplau et

al. [52] is able to detect improper use of dynamic code loading, it tasks the user with the responsibility of reusing the tool after every application update. Despite providing a way to detect suspicious behavior of an application, static analysis tools for Android do not help in mitigating the risk, unlike the tracer program proposed. It is also important to note that the majority of static analysis tools for Android operate just on Java bytecode and not on native code which is written in C or C++. Hence, a large attack surface is left unanalyzed [37].

8.2 Dynamic Analysis

Proprietary applications are often published using code obfuscation to prevent reverse engineering efforts. This creates an issue for static analysis techniques which analyze application bytecode. Therefore, monitoring an application while it is being executed offers better insights to tracking and countering the malicious behavior. Dynamic analysis tools such as DroidBox [51] and Androl4b [56] are able to overcome this drawback but these tools test the target application in a simulated environment which may or may not fully represent reality [58].

Framework modification can help runtime information flow tracking by using dynamic taint analysis [55]. Taint analysis is a technique wherein a software program is monitored to measure the effect of untrusted data sources on various functions performed by the program. TaintDroid [33] employs this to track sensitive data by marking it with a label (tainting the data). If the tainted data exits the system, the framework logs the data label, its destination and the application responsible for it. Since TaintDroid requires a framework modification, it raises problems during widespread deployment [36].

AppGuard developed by Backes et al. [25] extends the existing Android permission system by supporting stricter fine-grain security policies for applications. It avoids Android frame-

work modification by embedding the security monitor inside the untrusted application's code, thereby making AppGuard an inline reference monitor [34] unlike the tracer program proposed which acts as an external reference monitor. Aurasium developed by Xu et al. [60] enforces custom security policies by using `libc` interposition wherein it reroutes all invocation to native layer functions via its monitoring interface. This allows it to observe and modify all application and Android framework interaction similar to the tracer program. However, it relies on modifying the application bytecode to achieve `libc` interposition. Another application bytecode modification framework I-ARM-Droid developed by Davis et al. [30] achieves similar functionality by modifying an application's method calls. I-ARM-Droid not only depends on the user to identify the methods which require modification but also lacks a monitoring interface to check for any security violations. Injecting custom code in an application to enforce security policies during runtime requires repackaging and resigning of the application. This adds an additional burden on the user to download and modify the new version of the application every time an update is released. The tracer program proposed in Chapter 7 is able to enforce custom security policies without any framework modification or application repackaging and therefore tries to reduce the user's workload. If the target application is updated, the user would just need to update the security policy provided to the tracer.

8.3 Application Sandboxing

The work done by Bianchi et al. [27] also employs `ptrace`-based system call interposition to create a policy based application sandboxing environment termed NJAS. In their work, the target application is executed in the context of their sandboxing application NJAS. This allows NJAS to monitor and modify every operation performed by the target application.

To properly execute an application in the context of another application, engineering effort is required to identify important system calls relating to file system I/O and IPC binder communications that need to be intercepted and modified. Any failure in modifying these system calls will result in faulty execution of the application.

To avoid the trouble of executing the target application in its own context, the proposed tracer program primarily focuses on using system call interposition to enforce user-defined security policy and acts as an external reference monitor. Further, NJAS does not handle anti-debugging mechanisms which would prevent their sandbox from working correctly.

The sandboxing application Boxify proposed by Backes et al. [26] uses Android's *isolated process* feature wherein an application runs without any permission. The target application is granted specific permissions based on a custom security policy using a broker process and is run in the context of the Boxify application. This requires Boxify to reimplement major Android system services and broker all IPC binder communications. Unlike the tracer program, Boxify would require constant complex code modification to keep up with the newer versions of Android.

Chapter 9

Conclusions

Keeping in view the rising security threats, this thesis provides an analysis of the target UAS and explores potential sources of data exfiltration. With the DJI GO 4 application acting as the gateway to the Internet for DJI Spark quadcopter, an external monitor based on the man-in-the-middle technique is devised to monitor its network behavior. This investigation raises concerns warranting a fine-grain access control on Android applications. Procedures and recommendations are provided for drone operators in order to limit data leakage. Existing techniques to monitor data exfiltration for Android applications are discussed along with ways to prevent it.

Limitations of existing techniques and the Android security framework led to the development of the tracer program using system call interposition to monitor and modify system calls issued by a target application. The tracer program is evaluated against a simple weather application by providing it with a user-defined security policy. The tracer is further extended to monitor and modify system calls issued by the more complex DJI GO 4 application. Anti-debugging mechanisms to prevent attempts at reverse engineering DJI GO 4 are observed and methods to counter it are discussed.

With the majority of Android applications requesting network access, it is important to keep a constant check on them. Applications that have ties with authoritarian governments always present a risk of cyber-espionage [59]. Although the tracer program is designed to focus mainly on Android-hosted UAS applications, it is able to work with any Android

application available on the Google Play store. The tracer is not limited to monitoring the network operations of an application and could also be used to monitor any I/O operation such as file system access to prevent an application from exploiting the coarse-grain Android permissions.

9.1 Future Work

The tracer program has been implemented successfully and applied to work with a simple weather application. It is further extended to monitor DJI GO 4's network activity. However, several anti-debugging mechanisms were encountered while testing the tracer on DJI GO 4. In order to counter the anti-debugging mechanisms, workarounds using system call interposition to evade detection are discussed. However, further analysis is required to completely trace the DJI GO 4 application.

There is no generic way to bypass anti-debugging mechanisms but to better understand and analyze the defenses set by DJI GO 4, the application APK can be decompiled and studied by using tools such as APKtool [2] and Java decompiler [7]. This will help to develop better counter-approaches. Additionally, all the system calls executed during the load process of the application must be carefully observed to identify patterns which could help map the behavior of the application when the tracer is active.

Another way to analyze an application's behavior is by using dynamic binary instrumentation (DBI) frameworks. DBI frameworks analyze an application at runtime by injecting instrumented code [17]. This instrumented code gathers information about the process to help analyze the application behavior. Debuggers are able to achieve the same end by creating breakpoints as discussed in Section 5.3. However, DBI frameworks are immune to anti-debugging mechanisms. This is because the original code of the application is not exe-

cuted by DBI. A copy of the original code injected with the instrumented code is generated and executed [48]. Popular reverse engineering tools such as Frida [23] and Xposed [19] are a form of DBI framework used for hooking and code injection. They could be used to analyze DJI GO 4 and the observations could help in countering the anti-debugging mechanisms. If the constraints on application bytecode modification are relaxed then these reverse engineering tools could help in completely monitoring and modifying an application's behavior.

Different target applications would require different approaches to counter anti-debugging mechanisms. Testing the tracer on other Android-hosted UAS applications would help in adapting to the new complex ways in which these traps could be set. Further, the tracer could be packaged into a standalone application using the Java Native Interface (JNI). This would make it easier for the users to invoke the program directly from the smartphone while using any Android-hosted UAS application.

Bibliography

[1] Architecture | Android open source project. https://source.android.com/devices/architecture.

[2] Apktool- A tool for reverse engineering 3rd party, closed, binary Android apps. https://ibotpeaches.github.io/Apktool/.

[3] Application security | Android open source project. https://source.android.com/security/overview/app-security.

[4] Argus saf. http://pag.arguslab.org/argus-saf.

[5] DJI Enhances Software Security In Its Apps. https://www.dji.com/newsroom/news/dji-enhances-software-security-in-its-flight-control-apps.

[6] GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.

[7] Android apk decompiler. http://www.javadecompilers.com/apk.

[8] System and kernel security | Android open source project. https://source.android.com/security/overview/kernel-security.html.

[9] Permissions overview | Android Developers. https://developer.android.com/guide/topics/permissions/overview.

[10] Platform architecture | Android developers. https://developer.android.com/guide/platform.

[11] /proc. https://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html.

[12] ptrace(2) - Linux manual page. http://man7.org/linux/man-pages/man2/ptrace.2.html.

[13] Samsung Knox | Secure mobile platform and solutions. https://www.samsungknox.com/en.

[14] Security-Enhanced Linux in Android | Android open source project. https://source.android.com/security/selinux.

[15] System Calls (The GNU C Library). http://www.gnu.org/software/libc/manual/html_node/System-Calls.html.

[16] A Short History of Unmanned Aerial Vehicles. https://consortiq.com/media-centre/blog/short-history-unmanned-aerial-vehicles-uavs.

[17] Uninformed | DBI vol 7 article 1. http://uninformed.org/index.cgi?v=7&a=1&p=3.

[18] Wireshark. https://www.wireshark.org/.

[19] Xposed. https://repo.xposed.info/module/de.robv.android.xposed.installer.

[20] Seccomp filter in Android Oreo, Jul 2017. https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html.

[21] Fitness app Strava lights up staff at military bases, Jan 2018. https://www.bbc.com/news/technology-42853072.

[22] US warns of threat from Chinese drone companies, May 2019. https://www.bbc.com/news/technology-48352271.

[23] Frida, Jun 2019. https://www.frida.re/docs/android/.

[24] JP Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51 Air Force Electronic Systems Division (AFSC). *AD-758 206, ESD/AFSC.(Also available as Vol. I, DITCAD-758206. Vol. II, DITCAD-772806)*, 1972.

[25] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard - Real-time policy enforcement for third-party applications. 2012. https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/25262.

[26] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock Android. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 691–706, 2015.

[27] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. NJAS: Sandboxing unmodified applications in non-rooted devices running stock Android. *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices - SPSM 15*, 2015. doi: 10.1145/2808117.2808122.

[28] Blokadaorg. blokadaorg/blokada, Apr 2019. https://github.com/blokadaorg/blokada.

[29] czokie. Network Analysis Findings, Sep 2017. https://dji.retroroms.info/faq/dataleakage/chatter.

[30] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. I-arm-droid: A rewriting framework for in-app reference monitors for Android applications. *Mobile Security Technologies*, (2012):1–7, 2012.

[31] Dorneanu. dorneanu/smalisca, Mar 2017. https://github.com/dorneanu/smalisca.

[32] Edward J. Dumas and T. S. Wood. Network traffic study of a DJI S-1000 small un-

manned aircraft system (sUAS), Jan 2017. https://repository.library.noaa.gov/view/noaa/15960.

[33] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick Mcdaniel, and Anmol N. Sheth. Taintdroid. *ACM Transactions on Computer Systems*, 32(2):1–29, 2014. doi: 10.1145/2619091.

[34] Ulfar Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, 2004.

[35] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. *Trust and Trustworthy Computing Lecture Notes in Computer Science*, page 291–307, 2012. doi: 10.1007/978-3-642-30921-2_17.

[36] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. Understanding Android fragmentation with topic analysis of vendor-specific bugs. *2012 19th Working Conference on Reverse Engineering*, 2012. doi: 10.1109/wcre.2012.18.

[37] Stephan Heuser. *Towards modular and flexible access control on smart mobile devices*. PhD thesis, 2016.

[38] Jeremy Hsu. The Strava Heat Map Shows Even Militaries Can't Keep Secrets from Social Data, Apr 2018. https://www.wired.com/story/strava-heat-map-military-bases-fitness-trackers-privacy.

[39] Kapil Jain and R Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *NDSS*, 2000.

[40] Radhika Karandikar. Android application launch, Apr 2010. http://multi-core-dump.blogspot.com/2010/04/android-application-launch.html.

[41] Haye Kesteloo. Department of Defense bans the purchase of commercial-over-the-shelf UAS, Jun 2018. https://dronedj.com/2018/06/07/department-of-defense-bans-the-purchase-of-commercial-over-the-shelf-uas-including-dji-drones.

[42] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis - SOAP 14*, 2014. doi: 10.1145/2614628.2614633.

[43] Issie Lapowsky. How Cambridge Analytica Sparked the Great Privacy Awakening, Mar 2019. URL https://www.wired.com/story/cambridge-analytica-facebook-privacy-awakening/.

[44] Fiona Lau. Chinese drone maker DJI seeking at least $500 million in funds, Mar 2018. https://www.reuters.com/article/us-dji-tech-fundraising/chinese-drone-maker-dji-seeking-at-least-500-million-in-funds-sources-idUSKBN1GY0A7.

[45] M66B. M66B/XPrivacy, Jan 2018. https://github.com/M66B/XPrivacy.

[46] Maaaaz. maaaaz/androwarn, May 2019. https://github.com/maaaaz/androwarn/.

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

[48] Ncr. Anti-instrumentation techniques: I know you're there, Frida!, Nov 2015. https://crackinglandia.wordpress.com/2015/11/10/anti-instrumentation-techniques-i-know-youre-there-frida/.

[49] Lily Hay Newman. One of Most Popular Mac Apps Acts Like Spyware, Sep 2018. https://www.wired.com/story/adware-doctor-mac-app-store-spyware/.

[50] Ketan Parmar. In Depth: Android Package Manager and Package Installer - DZone Mobile, Oct 2018. https://dzone.com/articles/depth-android-package-manager.

[51] Pjlantz. pjlantz/droidbox, Oct 2017. https://github.com/pjlantz/droidbox.

[52] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute This! Analyzing unsafe and malicious dynamic code loading in Android applications. *Proceedings 2014 Network and Distributed System Security Symposium*, 2014. doi: 10.14722/ndss.2014.23328.

[53] Ben Popper. A government study found DJI drone, banned by US Army, kept data safe, Aug 2017. https://www.theverge.com/2017/8/7/16106810/dji-drone-banned-government-study-data-safety.

[54] Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, Inc., New York, NY, USA, 1996. ISBN 0-471-14966-7.

[55] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). *2010 IEEE Symposium on Security and Privacy*, 2010. doi: 10.1109/sp.2010.26.

[56] sh4hin. sh4hin/androl4b, Apr 2019. https://github.com/sh4hin/Androl4b.

[57] Tencent. Tencent/tinker, Feb 2019. https://github.com/Tencent/tinker.

[58] Bläsing Thomas, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An Android application sandbox system for suspicious software detection. *2010 5th International Conference on Malicious and Unwanted Software*, 2010. doi: 10.1109/malware.2010.5665792.

[59] Dustin Volz. Trump signs into law U.S. government ban on Kaspersky Lab software, Dec 2017. https://www.reuters.com/article/us-usa-cyber-kaspersky/trump-signs-into-law-u-s-government-ban-on-kaspersky-lab-software-idUSKBN1E62V4.

[60] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for Android applications. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 539–552, 2012.