POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Polytechnic Institute of Leiria
Higher School of Technology and Management
Department of Computer Engineering
Master in Cibersecurity and Digital Forensics

FORENSIC ANALYSIS OF THE GARMIN CONNECT ANDROID APPLICATION

FABIAN PEREIRA NUNES

Leiria, July of 2023

POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Polytechnic Institute of Leiria
Higher School of Technology and Management
Department of Computer Engineering
Master in Cibersecurity and Digital Forensics

FORENSIC ANALYSIS OF THE GARMIN CONNECT ANDROID APPLICATION

FABIAN PEREIRA NUNES
Number: 2210511

Carried out as part of Project under the guidance Professor Miguel Monteiro de Sousa Frade, Ph.D. and Professor Patrício Rodrigues Domingues, Ph.D..

Leiria, July of 2023

ACKNOWLEDGMENTS

I could not present this work without thanking everyone at my side since its beginning. Without them, it would have been impossible.

First, I want to thank my parents, the two most important people in my life. Without them, I would not even be here today. I want to thank them for all the love and support they gave me during this project and all my life. I also want to thank my sister and niece for their compassion and support.

I would have never completed this work without the mentoring and support from my advisors, Profesor Miguel Monteiro de Sousa Frade, Ph.D., and Professor Patrício Rodrigues Domingues, Ph.D., for all the knowledge they gave me not only here but also during their lectures.

Lastly, I wanted to thank the research center **Center for Innovative Care and Health Technology**, where I worked in the research **Safetrack** as a research fellow. In this project, we used Garmin Smartbands to monitor patients and send data to our software. I want to thank them because I would not have had the experience of working with Garmin without them, and I probably would not have had this idea for this work if not for them. I want to especially thank my research advisors, Professor Catarina Isabel Ferreira Viveiros Tavares dos Reis, Ph.D., and Professor Elga Patrícia Maximiano Ferreira, Ph.D. for their support and teachings, and lastly, the coordinator of the center and the research Professor Maria Pedro Sucena Guarino, Ph.D. for all the support and kindness given to me during my time as a research fellow.

ABSTRACT

Wearable smart devices are becoming more prevalent in our lives. These tiny devices read various health signals such as heart rate and pulse and also serve as companion devices that store sports activities and even their coordinates. This data is typically sent to the smartphone via a companion application installed. These applications hold a high forensic value because of the users' private information they store. They can be crucial in a criminal investigation to understand what happened or where that person was during a given period. They also need to guarantee that the data is secure and that the application is not vulnerable to any attack that can lead to data leaks.

The present work aims to do a complete forensic analysis of the companion application Garmin Connect for Android devices. We used a Garmin Smartband to generate data and test the application with a rooted Android device. This analysis is split into two parts. The first part will be a traditional *Post Mortem* analysis where we will present the application, data generation process, acquisition process, tools, and methodologies. Lastly, we analyzed the data extracted and studied what can be considered a forensic artifact. In the second part of this analysis, we performed a dynamic analysis. We used various offensive security techniques and methods to find vulnerabilities in the application code and network protocol to obtain data in transit.

Besides completing the Garmin Connect application analysis, we contributed various modules and new features for the tool Android Logs Events And Protobuf Parser (ALEAPP) to help forensic practitioners analyze the application and to improve the open-source digital forensics landscape. We also used this analysis as a blueprint to explore six other fitness applications that can receive data from Garmin Connect.

With this work, we could conclude that Garmin Connect stores a large quantity of private data in its device, making it of great importance in case of a forensic investigation. We also studied its robustness and could conclude that the application is not vulnerable to the tested scenarios. Nevertheless, we found a weakness in their communication methods that lets us obtain any data from the user even if it was not stored in the device. This fact increased its forensic importance even more.

INDEX

LIST OF FIGURES

LIST OF TABLES

ACRONYMS LIST

ADB	Android Debug Bridge.
AES	Advanced Encryption Standard.
AJAX	Asynchronous JavaScript and XML.
ALEAPP	Android Logs Events And Protobuf Parser.
API	Application Programming Interface.
APK	Android Application Package.
ART	Android Runtime.
AWS	Amazon Web Service.
BAT	Batch File.
BLE	Bluetooth Low Energy.
BLOB	Binary Large Object.
CBC	Cipher-Block Chaining.
CIA	Confidentiality Integrity Availability.
CORS	Cross Origin Resource Sharing.
CPU	Central Processing Unit.
CS	C Sharp.
CSRF	Cross Site Request Forgery.
cURL	Client URL.
CWE	Common Weakness Enumeration.

DFIR	Digital Forensics and Incident Response.
DLEAPP	Drones Logs Events And Protobuf Parser.
FIT	Flexible and Interoperable Data Transfer.
GC4AA	Garmin Connect for Android Analyzer.
GDPR	General Data Protection Regulation.
GMS	Google Mobile Services.
GPS	Global Positioning System.
GUI	Graphical User Interface.
HIPAA	Health Insurance Portability and Accountability Act.
HTML	Hypertext Markup Language.
HTTP	Hypertext Transfer Protocol.
HTTPS	Hypertext Transfer Protocol Secure.
IDE	Integrated Development Enviorment.
ILEAPP	Iphone Logs Events And Protobuf Parser.
IP	Internet Protocol.
JSON	JavaScript Object Notation.
JVM	Java Virtual Machine.
JWT	JSON Web Token.

KML	Keyhole Markup Language.
MITM	Man-in-the-Middle Attack.
MobSF	Mobile Security Framework.
NIST	National Institute of Standards and Technology.
OAuth	Open Authentication.
OS	Operating System.
OWASP	Open Web Application Security Project.
POSIX	Portable Operating System Interface.
RAM	Random Accessed Memory.
REST	Representational State Transfer.
ROM	Read Only Memory.
SDK	Software Development Kit.
SMF	Share Memory File.
SMS	Short Message Service.
SpO_2	Peripheral Capillary Oxygen Saturation.
SSL	Secure Socket Layer.
SSO	Single Sign On.

TSK	The Sleuth Kit.
UI	User Interface.
URL	Uniform Resource Locator.
USB	Universal Serial Bus.
UTC	Coordinated Universal Time.
VLEAPP	Vehicles Logs Events And Protobuf Parser.
VM	Virtual Machine.
vO_2	Maximal Oxygen Consumption.
VPN	Virtual Private Network.
WAL	Write-Ahead-Log.
WLEAPP	Windows Logs Events And Protobuf Parser.
XML	Extensible Markup Language.
XSS	Cross-site scripting.

1

INTRODUCTION

We live in the golden age of technology, and everyday researchers and engineers are developing new ways to integrate technologies into our lives and make them available to more people. In the last two decades, our computers have gradually transformed from big machines to portable devices that we carry in our pockets or, in this case, our wrists. This project will focus on one of the popular devices called smart wearables in specific smartbands and smartwatches.

These tiny wrist devices can already function as smartphones by executing advanced functions like running apps, placing phone calls, having Global Positioning System (GPS), and storing information. However, smartbands and smartwatches are two very different kinds of devices.

A smartwatch mimics a traditional watch. However, it does much more than show us the current time. A smartwatch runs a custom-made Portable Operating System Interface (POSIX) Operating System that requires fewer resources and functionalities (WearOS based on Android and WatchOS based on IOS). This Operating System (OS) gives the watch many capabilities of a typical smartphone, like running apps and making phone calls (if it has cellular capabilities). However, one of the most important aspects of a smartwatch is the various sensors that can read values such as Hearth Rate Variation and Blood Pressure. These sensors turn the watch into a fitness and health tracker. One of the biggest markets for this kind of device are people doing workouts or wanting to monitor their health constantly. The Apple Watch, for example, has already been proven to help diagnose heart problems and cancer using its monitoring system. The University of Cincinnati joined research to help prevent strokes with the Apple Watch (Bangert, 2022). Image 1 shows an example of a typical smartwatch, in this case, Garmin Venus 2.

A smartband is a more specific device focusing on the fitness aspect of the smartwatch, and they usually run a very basic OS with minimal resources to have an interactive interface. These devices have minimal capabilities. Their goal is to work as a fitness tracker since they also have various health-related sensors. Unlike a smartwatch, a smartband does not receive calls or send messages and cannot run applications. The goal is to monitor daily physical activities and fitness-

Figure 1: Garmin Venus 2

related metrics such as steps, running distance, heart rate, sleep patterns, swimming laps, calories burned, and more. Usually, these devices possess minimal storage capacity, retaining captured data for a restricted duration until the user syncs it with the companion application. Instead, it communicates (usually using the protocol Bluetooth Low Energy (BLE)) with a companion app on the user's device. This application stores the users' metrics and workout sessions. Figure 2 shows an example of the smartband used in this project, the Garmin Vivosmart 4. These applications work with smartwatches and smartbands. The difference between these devices is that the smartwatch manages to store information on the device so it can work independently from the application. Bands do not give the user access to the files stored inside and are usually stored in specific file formats. The smartband is more dependant on the companion application than a smartwatch.

1.1 MOTIVATION

Because of the sensors inside these devices, they gained much attention in the last years in digital health and fitness. They are a core element in creating applications and projects to read users' health metrics and current location. In 2019, more than 350 000 healthcare applications were already published in the major app stores, responsible for 3.7 billion downloads per year (Byambasuren et al., 2019). In 2022 there have been sold more than 68 million devices the key brands are:

1. Apple

Figure 2: Garmin Vivosmart 4

2. Samsung
3. Garmin
4. Fitbit
5. Fossil

With the increasing popularity of these devices also come security-related questions. It is nothing new that every year there has been an increase in attacks and security breaches. One of the most affected areas is the mobile application market. A study made by Technologies (2019) found out that in 17 applications studied by the organization, almost 50% had high-risk vulnerabilities. The author explains in a few topics some of the reasons caused during development that make applications vulnerable and create exploits:

- Poor or no data protection and encryption
- Use of outdated components and third-party libraries
- Third-party information sharing
- Lack of secure code by design

Mobile Healthcare and fitness apps are one of the dominant applications in the current market. Various studies have been made in this area, analyzing the current threat landscape and security challenges these applications face. The most notorious was the research **Analyzing security issues of android mobile health and medical applications** that used a platform to test 20 000 mHealth applications discovering several important factors (Tangari et al., 2021):

- mHealth apps generally adopt more reliable signing mechanisms and request fewer dangerous permissions than other applications.
- 1.8% of mHealth apps package suspicious codes, and 45% rely on unencrypted communication. As much as 23% of personal data (location information and passwords) transmits on unsecured traffic.

That is why applications like companion apps must have enough protection to prevent attackers from obtaining their stored data since they hold a wealth of private information from the user, like phone numbers, health records, and location.

This project will focus on elaborating a complete mobile digital forensics analysis of one of the most used companion applications in the world, Garmin Connect (similarweb, 2023), to discover what data it stores and how secure it is. Mobile Forensics comprises a subdivision of digital forensic analysis that gathers traces and extracts relevant information from a mobile device. Typically the researchers follow good practices and international standards to preserve the integrity and authenticity of the information so that it is admissible in court and the chain of custody is kept. Since these applications store health-related values (for example, heart rate and stress level) and share the user's location at a given time, they can be instrumental in digital forensics and have already helped solve many cases. An example of using a smartwatch to solve a crime was a murder victim's Fitbit data used to prove that her husband had committed the crime. The Fitbit timeline proved movement within the house and distance traveled, placing him at the crime scene contrary to his testimony (Watts, 2017). Another example is how the GPS coordinates of a Garmin Smartwatch and its companion application helped police convict a man of two murders (Shweta, 2019).

To conclude, this project aims to use various forensic and security tools and mechanisms to make a complete forensic analysis of the companion application Garmin Connect for Android that tracks the user activities and records captured with Garmin wearables. This report will show the complete setup and process executed, the data found, and the analysis results. These data will benefit digital forensic practitioners, that will know what data are expected to be gathered when analyzing a device with the Garmin application installed.

1.2 OBJECTIVES

This analysis aims to acquire all potential forensic artifacts produced and stored by the Garmin Connect application while interacting with the Garmin Vivosmart 4 band. We aim to determine whether these artifacts raise privacy concerns for the end user and assess whether the application developers have implemented adequate security measures to safeguard the data against potential attackers.

Another goal is that by using open-source tools for forensic analysis, other analysts can use this project as a blueprint for future research and bring a more prominent focus to studying applications associated with wearables.

Lastly, we aim to develop modules related to Garmin Connect for the forensic tool Android Logs Events And Protobuf Parser (ALEAPP). This tool is a file parser that extracts forensic artifacts from files and presents them in a generated report, ALEAPP is integrated into the forensic tool Autopsy.

1.3 CONTRIBUTIONS

The main contribution is to the digital forensics community by providing a thorough analysis of a fitness application since, as discussed in the next chapter, forensic analysis of fitness applications involving wearables is still basic, as these analyses focus only on one or two artifacts and only search thoroughly for some pieces of data. Our goal is for this analysis to be used as a reference for future analysis of this application. We also hope that the data found here can be used by police forces when investigating the Garmin Connect application when connected to a possible crime scene, such as the Fitbit example given before.

By analyzing Garmin Connect, we used the application as a case study that could be adapted to other applications of the same genre. To prove this, we used the methods implemented there to analyze six other fitness applications.

Another significant contribution of this project is the development of 36 open-source plugins for the well-known forensic tool ALEAPP (for Garmin Connect and other applications), six new features for it, and five open-source scripts that will help analysts in their line of work.

In our dynamic analysis of the application, we thoroughly examined the application's Application Programming Interface (API) and successfully accessed user data using the obtained token during the communication process. Our methodology can be a foundation for other analysts aiming to conduct real-time forensic analyses of similar applications.

In short, this project will bring as contribute:

- Full documentation of a forensic analysis of Garmin Connect for Android.
- Share the knowledge gained with users, analysts, and the company.
- Creation of various open-source tools for Android forensic analysis (repositories shown in Table 31).
- Creation of various features and plugins for the open-source tool ALEAPP (repository shown in Table 31).

Lastly, the work conducted during this research project has led to the development of a paper titled **"Post-mortem digital forensic analysis of the Garmin Connect application for Android"** which is currently submitted to the journal **Forensic Science International: Digital Investigation**.

1.4 STRUCTURE

The remainder of this project is organized as follows. Chapter 2 reviews related work, while Chapter 3 analyzes the Garmin Connect application, highlighting its primary forensic artifacts. Chapter 4 explores the Garmin Connect application using various dynamic tools to find vulnerabilities and get data. Chapter 5 presents our open-source modules, Garmin Connect for Android Analyzer. Chapter 6 analyzes five fitness application in a *post-mortem* scenario using the methods applied to Garmin Connect. Finally, Chapter 7 concludes this study.

2 RELATED WORK

Forensic investigations for wearables and their companion applications are nothing new. Before beginning this project, the authors did extensive research to try and find various kinds of research that focus on obtaining data from these devices and applications. By studying these various research, the authors wanted to understand better how other researchers studied companion applications and what tools and methods they used. This project aims to innovate and solve potential problems that the other researchers had and, most importantly, if there had already been studies about the Garmin Connect application. In the case of existing studies about Garmin, we will explain their shortcomings and where we could innovate. Luckily for us, compared to other applications described here, there have been few types of research done on the Garmin Connect application, and there is still more to explore, as will be shown during this project.

During this research, we found various types of investigations into wearables that focused on different things and used different methods. That is why we decided to split this into four types of research:

1. Data acquisition from the device and desktop application
2. Data acquisition from the mobile application using static methods
3. Data acquisition from the mobile application using dynamic methods
4. Use and development of ALEAPP in investigations

This project will not focus on data acquisition directly from the smartband. However, it is vital to discuss this type of investigation when discussing wearables.

2.1 DATA ACQUISITION FROM THE DEVICE AND DESKTOP APPLICATION

Acquiring data from a smartwatch is similar to a smartphone since they share a similar file structure. Researchers often use open-source or paid tools to make an image from the device's persistent memory and then analyze it in a forensic platform to search for artifacts like personal information and delete data.

Table 1: Artifacts by Almogbil et al. (2020)

BRAND	FINDINGS
Fitbit	User information
	Device Information
	Group Information
	Pictures
	Friends
	Activities
	Messages
	Community Feed
	Payment Information

The first research found was from Almogbil et al. (2020), where they used two different models of the Fitbit bands during a specific period. After that, they imported the data to the Fitbit Windows 10 desktop application and analyzed it with the digital forensic Autopsy software from the application finding several private information shown in Table 1.

MacDermott et al. (2019) analyzed the data stored on three different devices: a **Garmin Forerunner 110**, a **Fitbit Charge HR**, and a **HETP** tracker (generic brand). The authors used Autopsy and FTK Imager tools to obtain and study the files and the software FitSDK and GoldenCheetah (software for analyzing Fitbit metrics) to open the Fitbit proprietary files. The research only identified a limited number of data. The researchers made a copy of the device's data for the Garmin watch and analyzed it with Autopsy, finding artifacts related to past activities. For the Fitbit band, the authors analyzed the desktop application for Fitbit using the forensic tool Autopsy just like the researchers Almogbil et al. (2020), finding several forensic artifacts such as personal information and user-related workouts. However, comparing both studies, MacDermott et al. (2019) found fewer data in the Fitbit application. The authors found the artifacts listed in Table 2.

Lastly, a study by Kim et al. (2022) tried to use different methods to obtain the data stored in a smartwatch. They chose three devices – **Galaxy Watch 3**, **Apple Watch 5**, and **Garmin Vivosport**. They used methods from the least invasive – connecting directly to the computer – to the most invasive – performing chip-off. Using these techniques, they found a wealth of information stored in the memory of these devices. They obtained the artifacts listed in Table 3.

Table 2: Artifacts by MacDermott et al. (2019)

BRAND	FINDINGS
Garmin Forerunner 110	User workouts
Fitbit Charge HR	User information User workouts GPS location
HETP	No information was extracted

Table 3: Artifacts by Kim et al. (2022)

BRAND	FINDINGS
Samsung Galaxy Watch 3	Device Information Connected Smartphone Information Text/Voice Messages Contacts Calendar Location App logs Health Records User Credential
Apple Watch 5	Bluetooth Information MAC Information Installed App list Contacts Mail Address Media File List
Garmin Vivosmart	Exercise Log Hearth Rate Log Connection Log Settings Log

Table 4: Artifacts by Yoon and Karabiyik (2020)

BRAND	FINDINGS
Fitbit Versa 2	GPS Location Health Values App ID Web cookies Credit Card Info and Image Alexa Serial Number

2.2 DATA ACQUISITION FROM THE MOBILE APPLICATION USING STATIC METHODS

Most studies found digital forensic artifacts in wearables, almost exclusively focusing on static methods to analyze the device and the companion application. Our work considers both static and dynamic analysis. Static methods focus on analyzing files and data after a given application usage. Almost all the research in this section will analyze artifacts in the application's database and configuration files, often Extensible Markup Language (XML) or JavaScript Object Notation (JSON) files.

Hassenfeldt et al. (2019) studied nine different fitness applications for Android: MapMyFitness, RunKeeper, Strava, MyFitnessPal, Runtastic, Health Infinity, Fitness Tracker, Nike Training, and JEFIT. The authors created their testing environment by collecting and extracting data with the tool, Android Debug Bridge (ADB). Their main findings were `Personal Data`, `GPS location`, and `Passwords` related to the applications. The authors also developed a tool for extracting forensic artifacts, although not as complete as resorting to the ALEAPP framework.

Yoon and Karabiyik (2020) published a forensic study of the Fitbit Versa 2 for Android. The research explains the triage process one must follow in investigating wearable devices. How the device should be apprehended, and how the data needs to be acquired. The researchers used static methods and commercial tools such as **AXIOM** and **XRY** to acquire data from the device and study it afterward. The authors found many relevant forensic artifacts inside the `SQLite3` databases shown in Table 4.

Kang et al. (2020) studied the **Fitbit Alta HR** and the **Xiaomi Mi Band 2** and their respective Android application, focusing on the forensic artifacts found in the `SQLite` databases from the applications. The authors reported user-related information that the applications store, like sleep, steps, activities, account, and device information.

Williams et al. (2021) reported the methods used to acquire data from the Fitbit application on Android and IOS. The authors studied whether the retrieved data differed between the two operating systems. To test this, the authors created two scenarios, one using a Google Pixel 2XL and the other with iPhone 7 Plus. They used two commercial forensic tools – **Cellebrite** and **XRY** – to extract and study the data on the computer. Since the Android device was not rooted, the tools could not extract the private information of the application, so the authors used a virtual device created with the **Genymotion** emulator. The data found was the same as the other research made for the Fitbit application, such as **Private Messages**, **Feed Posts**, **GPS Data**, **Profile Information**, **Sleep Data**, **Heart Rate Data**. This research highlights the difference in the acquisition methods of both operating systems.

In 2021 Dawson and Akinbi (2021) analyzed the contents of the Tom Tom companion application focusing on the data stored in the TomTom Spark 3 watch. The author's goal was to compare the forensic artifacts found in the TomTom watch using forensic and non-forensics tools and demonstrate the possible limitations of these tools and how they can affect the analyst's decision-making. To that end, the authors compared the data obtained with the Cellebrite forensic tool with those studied using ttwatch [1] – an open-source command-line tool used to interact with the physical TomTom GPS smartwatch and extract forensic artifacts stored on flash memory – and Runanlyze [2] – a tool to analyze proprietary files from TomTom watches. The authors found forensic data related to **Activities**, **User Account**, and **Bluetooth Logs**.

Domingues et al. (2023) did a *post-mortem* analysis of the companion application ZeppLife (formerly called MiFit) for Xiaomi devices when coupled to a MiBand 6 in a rooted smartphone. The authors focused solely on a static analysis of the application, reporting on the following data:

- Health data
- Device Data
- Daily summaries (steps, sleep hours, etc.)
- User information

1 https://github.com/ryanbinns/ttwatch
2 https://runalyze.com/

- Workouts

They also developed a software module – **MiFit Analyzer** – for the Autopsy forensic browser. The module generates a dynamic HTML-based report with the artifacts found in the extracted private directory of the application. Although we also resorted to a rooted smartphone, our analysis of the Garmin Connect provides both a static analysis of the data generated by the application and a dynamic study of the data generated during the application's execution, while Domingues et al. (2023) focused solely on *post-mortem* data.

Hutchinson et al. (2022) studied three companion applications using three different smartwatches and smartbands, the **Amazon Halo Band**, the **Garmin Vivosmart 4**, and the **Mobvoi TicWatchS2**. This research is different from the rest. One of the applications is Garmin Connect for Android, using the same smartband as we are, the Garmin Vivosmart 4. The authors created a test environment using the various smartbands to populate the application's database. They used a rooted Samsung A50 with Android 10 to facilitate the post-acquisition of the data. After that, they analyzed the contents of the application using three different tools, **Cellebrite** and **Magnet Axiom**, both commercial tools and the popular open-source tool Autopsy. The authors aimed to find the differences using commercial and open-source tools in *post-mortem* examinations. They also explored various other types of research made before them in specific research about Fitbit. The authors found the data in Table 5.

Regarding digital forensic data, the authors focused mostly on the application XML files, devoting less attention to the application's database. Our work provides a deeper analysis of the *post-mortem* data left by the application regarding the databases. In addition, we also investigate the dynamic behavior of the application and provide for software modules to report on the digital forensic artifacts left by the application usage.

2.3 DATA ACQUISITION FROM THE MOBILE APPLICATION USING DYNAMIC METHODS

Our work includes the analysis of the dynamics of the application, that is, the code structure and the communication with the cloud. Thus, we review work focusing on dynamic analysis from a digital forensics perspective.

Table 5: Artifacts by Hutchinson et al. (2022)

BRAND	FINDINGS
Amazon Halo	Profile Information Heart Rate Data Steps Data Exercise Data Sleep Data Voice Data
Garmin Vivosmart 4	Profile Information Steps Data Exercise Data Sleep Data Stress Data App Notifications
Mobvoi	Profile Information Heart Rate Data Steps Data Exercise Data Sleep Data

The institute AVTest released an article evaluating the security of nine different fitness trackers performed by Schiefer (2015). The brands studied were Acer, Fitbit, Garmin, Huawei, Jawbone, LG, Polar, Sony, and Withings. The research focused on analyzing the communication between the smartband and companion application and between the application and the cloud. The authors also analyzed the application's code to find possible flaws. One of the critical points of this research was intercepting the Bluetooth communication between the device and phone and modifying the data before it reached the application. They found out that:

- User needs to authenticate to transmit data
- The code is not obfuscated
- There are no log entries
- Sensitive information is stored in the private directory
- Communication is encrypted

Another research in the field was from Fereidooni et al. (2017). The authors used various offensive techniques to try and test the overall security of the communication between 17 different fitness trackers and their respective companion applications, including wearable from Garmin and the Garmin Connect application. For this purpose, the authors created a testing environment using the fitness tracker and the application. They put a computer running a man-in-the-middle proxy to receive the communication packets before the application sends them to the remote server. The process used in this research is complex and faces various challenges, such as:

- Data Encryption
- Proprietary Encoding
- Data Integrity Check
- Secure Socket Layer (SSL) certificate pinning

The authors found that the communication uses the Hypertext Transfer Protocol Secure (HTTPS) protocol. However, the data is encoded in a proprietary protocol, not encrypted. The authors created a script in Perl that receives the data file in transit, modifies it, and sends it instead of the original to the server. The authors successfully uploaded 80 million steps to the Garmin server, which should be impossible. The authors used the same process for the other devices, with changes depending on the data protocol and encryption.

Since we could not find more research about wearables and dynamic analysis, we searched for other researchers that did a forensic analysis of the mobile application with dynamic methods to test the application. The goal was to learn about possible tools to use on this project and to see how they did it and what data they could acquire. It was possible to find more research here, but we decided to specify one research.

Barros et al. (2022) published a forensic analysis of the Bumble dating application. The application is unrelated to our project, but the methodology is relevant in static and dynamic space. Specifically, the authors used various methods and tools that fit our project, such as Man-in-the-Middle Attack (MITM), certificate pinning, and a Virtual Private Network (VPN) to obtain data in transit between the Android application and the cloud. They also developed a script in Python to generate a document with the chat messages sent and received by the user. Even if the application is not essential for this project, the static and dynamic analysis techniques will prove helpful here.

2.4 USE AND DEVELOPMENT OF ALEAPP IN INVESTIGATIONS

As our modules target this framework, we also analyzed works related to ALEAPP. ALEAPP is a popular open-source Python-based framework able to extract forensic artifacts from an application's data folder and create reports (Brignoni, 2023) through specific software modules. The modular framework allows developers to add new modules supported by existing features of ALEAPP and develop new functionalities to enhance the tool's report capabilities. In this short review, we focus on:

- Works that have developed modules for the framework
- Analysts that have used it in their studies

The work by Vasilaras et al. (2022) resorted to ALEAPP to analyze the `SQLite` database files of the Android Telegram application. Analysts Delija et al. (2022) relied on ALEAPP, Autopsy, and the commercial tool Belkasoft [3] to process forensic artifacts found in the system files of Android version 11 to compare the results provided by the three tools. Lastly, Mirza et al. (2022) performed a digital forensic analysis of various Web3 wallet applications for Android and iOS. In their research, they used both ALEAPP and its similar platform for iOS called Iphone Logs Events

3 https://www.google.com/search?client=firefox-b-d&q=belkasoft

And Protobuf Parser (ILEAPP). Our work also relies on ALEAPP, as we provide modules to process the forensic data of the Garmin Connect application. We have also extended the framework, adding new features and capabilities such as Heatmaps, Date Filtering, GPS Maps, and Data Charts, as we shall see later in chapter 5.

Other tools derived from ALEAPP currently serve the same purpose but for different systems. In chapter 5 section 5.7, we explain in detail the difference between this tool and what we call the **LEAPP Ecosystem**.

2.5 PRIVACY FITNESS WEARABLE

Lastly, we found another research about privacy and the consequences of a possible attack on a fitness tracker application, even if it is not about forensic research. The research done by Saha et al. (2017) focused on the privacy challenges companion applications suffer and how to mitigate them. The research identified various current problems, such as:

- User Identification
- Leaking of Medical Condition and other confidential data
- User tracking based on the GPS data

It also identified the most prominent type of attack, MITM, and the method to solve this problem is by applying the Confidentiality Integrity Availability (CIA) triad. The author even used as an example the Garmin connect application that, while writing the research, still used Open Authentication (OAuth) 1.0 for their authentication, a legacy protocol already replaced by its 2.0 version (okta, 2023), that addressed various security flaws. This paper does not add much forensic value. Still, it shows the other side of the coin, where security is essential and the damage that security failure can cause to the end user.

Since this project will use a fitness tracker, we will only focus on the data stored by the companion application. We will use various tools and methods implemented by the other researchers in the static analysis. We also hope to innovate by using different tools and even creating tools to help further researchers in their works. The same will happen for the dynamic analysis, which is more uncommon in the research about the companion application and the tools used. The result will depend on the security mechanism implemented by Garmin Connect.

3 POST MORTEM ANALYSIS OF GARMIN CONNECT

In this chapter, the authors will detail all the necessary steps to do a complete *Post Mortem* analysis of the Garmin Connect application, first by studying the application and using it, gathering information, and then extracting and analyzing the data generated by the application by using various tools and methodologies.

3.1 GARMIN CONNECT

Garmin is an American Company that primarily develops hardware and software related to GPS technology (*Garmin Portugal* 2023). Garmin is also one of the most well-known brands in the wearables market, having various smartbands and smartwatches in different price ranges focusing on fitness and well-being.

The mobile companion app used by Garmin Wearables is called Garmin Connect. Its primary focus is to serve as a fitness tracking application storing all the data related to the users' exercise and helping monitor their health and performance parameters when this application receives data from Garmin devices.

The main features offered by Garmin Connect are the following:

- View daily health data and statistics in detail in the application's dashboard.
- Analyze activities and their related statistics.
- Create customized workouts.
- Synchronize with other fitness applications such as MyFitnessPal and Strava
- Review personal records for steps, distance, and pace.
- Earn and share with friends accomplishments for challenges.
- Measure up to other Connect users with Insights.
- Get support for Garmin devices and their features.

Our motivation to study the Garmin Connect application stems from providing a thorough analysis of the application from a digital forensic point of view, both at

Table 6: Studied application details

App name	Garmin Connect
Downloads	+10 million
Average user score	4.6 / 5
Number of reviews	+800 000
Market	Global
Studied version	4.61 (released on 2022-11-15)
Play store URL	https://play.google.com/store/apps/details?id=com.garmin.android.apps.connectmobile

the static and dynamic levels. Garmin is gaining popularity in the wearable market, ranking fifth among the most sold brands and dominating the premium watch market (Lovejoy, 2022). On Google Play, the Android Garmin Connect application has surpassed 10 million downloads and has a 4.6 score out of 5 from more than 800 000 reviews (more details are shown in Table 6).

The Garmin Connect application requires a set of Android permissions that must be enabled when the application executes for the first time. Table 7 lists the permissions asked by the application. All permissions made sense based on the features provided by Garmin Connect and compared to other applications in the same field. Installing the application for the first time will ask permission to access the calendar, the user's location, calls, and messages.

The developers also have a section related to data privacy in the store. This section is divided into three parts: **Data Sharing**, **Data Collection**, and **Security Practices**. Next, we briefly review each of these parts.

data sharing The developers affirm that they do not share data with any third-party user or application without prior consent from the user.

data collection In Table 8, we show the data shared by the application according to Garmin [1].

1 https://play.google.com/store/apps/datasafety?id=com.garmin.android.apps.connectmobile

Table 7: Garmin Connect Permissions

PERMISSION	FUNCTION
Calendar	Read calendar details and events
Camera	Take pictures and videos
Contacts	Read Contacts
	Search for accounts on the device
Location	Only access the approximate location in the foreground
	Only access the exact location in the foreground
Call Logs	Read Call Log
Cell Phone	Dial phone numbers directly
	Read the status and identity of the mobile phone SMS
	Send and view SMS messages
Storage	Change/delete the contents of the shared storage
Other	Transfer files without notification
	Prevent phone sleep mode
	View Wi-Fi connections
	Receive data from the internet
	Use background data
	Access background location
	Send fixed broadcast
	Have full access to the network
	Run in background
	Advertising ID authorization
	Run service in foreground
	Access Bluetooth settings
	This app may appear on top of other apps
	Run on startup
	See network connections
	Read Google services configuration
	Query all packages
	Control vibration
	Answer phone calls
	Installer Referrer API Google Play
	Sync with Bluetooth device

Table 8: Data Shared by Garmin Connect

TYPE	INFORMATION
Personal Information	Name Email User ID Other Information
Health and Fitness	Health Information Fitness Information
Contacts	Contacts
App performance and information	App performance and information

security practices The developers detail some of the security features they have implemented for the application to be compatible with the current privacy and security laws worldwide.

- Encrypting all the data that is in transit
- Users can ask for their data to be deleted

3.2 STATIC ANALYSIS

In this section, we will detail the process of analyzing the application, the tools used, and what we found.

To realize this analysis, we will use as a guide the various research studied in Chapter 2, but also three excellent books for mobile forensics. The first, called **Practical Mobile Forensics** (Tamma, 2020), explains in detail how to realize a forensic investigation on Android, IOS, and Windows devices. The second book – **Learning Android Forensics** (Skulkin et al., 2018) – focuses exclusively on Android, and it's a hands-on book demonstrating various tools and techniques to extract data from Android devices. Lastly, **Learning Python for Forensics** (Miller and Bryce, 2016) teaches various methods and tools in Python for developing scripts that can prove helpful in a forensic investigation. Furthermore, we will reference other publications during this chapter when we use a tool discovered in one of them.

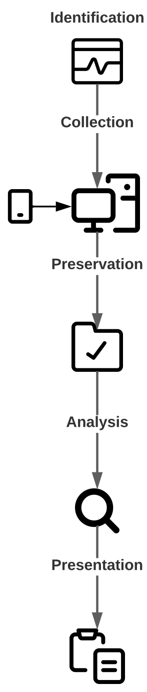


Figure 3: Method proposed by NIST for Digital Forensics (Kent et al., 2006)

3.2.1 *Concept and technologies*

First, it is essential to understand the meaning of static analysis or post-mortem analysis. In the field of digital forensics, static analysis refers to the copy of the data of a device or application and posterior analysis.

Here the author will install the application on an Android phone and use it to generate data. Then we will extract the data from the mobile phone to their workstation and analyze the individual files. This method is the standard approach in mobile application forensics since it is a relatively simple and consistent method of analyzing data. Developers even use these techniques to know what data their application stores on the device and if it is secure. However, more applications store the bare minimum on the device, resorting instead to the cloud. Thus analysis cannot focus only on analyzing the local files stored in the machine and need to use different tools and techniques to obtain data during the application's runtime (also called dynamic analysis).

There are various documented ways of realizing digital forensic investigations. The most commonly used in mobile forensics are the methods proposed by NIST in their publication 800-86 (Kent et al., 2006) that integrated forensic techniques in incident response plans as shown in Figure 3.

This process was designed for static analysis since it follows all the steps shown in Figure 3, which is why books like Practical Mobile Forensics use this diagram to realize mobile forensics. Here the analyst follows these steps:

1. Identify all relevant features of the application and use them to generate data
2. Collect the data using forensic tools or manual methods from a rooted phone or emulator
3. Preserve the data in a secure location and make sure the data is not altered or tampered
4. Examine the data using forensic or non-forensic tools and methods
5. Analyse the respective findings
6. Report and present the results of the investigation

To analyze the Garmin Connect application, we will use the method proposed by NIST.

3.2.2 *Tools*

In this subsection, we will detail all tools needed for static analysis of the Garmin Connect application. This project focuses on using free and open-source tools and developing software whenever needed.

The static analysis is split into two parts. The first was obtaining the data from the device and manually analyzing it. After that, we analyze the application using an automated mobile security assessment framework.

As stated before, we used ADB and our script **ADB-Extractor** for extracting the user data from the device.

For analyzing databases, we resorted to the open-source tool **DB Browser for SQLite** [2], which is one of the most popular tools for working with SQLite databases. SQLite is currently the defacto database choice for mobile developers. We will explain it in greater detail in subsection 3.3.3. Other database-related tools include **schemacrawler** [3] to generate database diagrams to understand relationships between tables, and **DBDiagram.io** [4] to improve the diagrams generated by schemacrawler. To attempt to recover deleted records from SQLite databases, we

2 `https://sqlitebrowser.org/`
3 `https://www.schemacrawler.com/weak-associations.html`
4 `https://dbdiagram.io/home`

Table 9: Static Analysis Tool

TOOL	VERSION	USE
ADB	33.0.1	Data access
bring2lite	1.0	Recover deleted records
DB Diagram	online tool	Create database diagrams
DB Browser for SQLite	3.12.2	Database analysis and queries
schemacrawler	16.19.5	Generate database diagrams
MobSF	3.6.2	Automated, mobile security assessment framework

used the open-source script **bring2lite** [5]. We also needed to analyze the text files inside the folders.

In the second part of the analysis, we used the automated tool Mobile Security Framework (MobSF) [6]. MobSF is one of the most advanced open-source tools commonly used by vulnerability analysts. MobSF is constantly updated and compatible with security frameworks such as Open Web Application Security Project (OWASP). Not only that, but it is one of the few maintained tools dedicated to mobile security. We present all the tools, versions, and use cases in Table 9.

3.2.3 *Methodologies*

In this subsection, the authors explain the methods used for testing and generating data in the application and extracting the data.

IDENTIFICATION The first step is, of course, to install the application on an Android device. For that, one must use a rooted phone or emulator because that is the only way of extracting all the data from it. A rooted phone is usually preferred, as the Android emulator cannot run certain applications. This is the case with Garmin Connect, as we need a real device to connect to the smartband.

The Android operating system is POSIX based, and such as desktop operating systems, it is permission based. However, in Android, the user does not have access to the administrator privileges like in a typical POSIX system using the command

5 `https://github.com/bring2lite/bring2lite`
6 `https://github.com/MobSF/Mobile-Security-Framework-MobSF`

Table 10: Samsung A40 Specification

SPECIFICATION	INFORMATION
Brand	Samsung
Model	A40
Android Version	11
Storage	64GB
RAM	4GB
Chip	Exynos 7904
Launch Date	19/03/2019

`sudo`, meaning that one does not have access to specific directories or commands to execute via a command line. That is why rooting the device is so popular among Android devices. Rooting allows users to attain higher administrative privileged controls. Rooting an Android device gives users the necessary privileges to install unapproved apps, update the operating system, delete unwanted system applications, under-clock or overclock the processor, replace the firmware, and access private directories of apps.

The rooting process is becoming more difficult as brands create ways of preventing this process altogether, making this process hard and dangerous as the devices can be bricked (Unuchek, 2017). Currently, the brands that make this process less difficult are: **Google** and **Oneplus** (Hildenbrand, 2023). The user must follow a detailed list of steps to ensure the root is successful. If not, he/she can completely brick the device, making it unusable. Rooting also voids the device of any guarantee. In order to conduct a comprehensive analysis, it is essential to obtain access to the private data stored by the application, which necessitates rooting the phone. However, in a real scenario, this approach may not be feasible as it would result in formatting the phone and subsequently losing valuable evidence. Many developers are creating ways of detecting emulators and rooted devices and preventing the application from executing making the analyst's job more difficult since they need to find ways to bypass these restrictions.

For this analysis, we used an **Samsung A40**. Table 10 shows the phone's main specifications.

The rooting process is split into various steps. The two main ones are the installation and configuration of **Magisk** [7] and **Odin** [8].

Magisk is a root and universal system-less interface developed by **XDA**, free and open source for Android devices running on version 4.2 and later. Magisk roots the device using a boot image patching method that essentially allows full root access to the Android OS without altering or modifying the /system partition (systemless root). Magisk also includes a feature known as Magisk Hide, which hides root from apps that search for root. Various applications claim to root a phone by simply installing it from the play store, such as **SuperSU** and **KingRoot**. Over the years, they have proven unreliable and do not offer an authentic root experience. Magisk is one of the only tested and maintained methods of securely rooting an Android device.

Another necessary tool for rooting a Samsung device is Odin, which is a custom `ROM` flashing tool for Samsung smartphones and tablets.

We followed a guide to install both magisk [9] and flash the Read Only Memory (ROM) with ODIN [10], completing the boot process successfully. This process depends on the brand, model, and phone version, and many things can go wrong, so the user should know how this process works.

After setting up the phone, we installed the Garmin Connect application from the play store. After installing it, the application's first screen is the landing page, where the user must sign up or log in with an existing Garmin account. The signup process was as follows:

1. Create a Garmin account with the user's name, email, and password, choose their country, and accept that they have at least 16 years (this conflicts with the age rating in the store page) and optionally, if they want to receive publicity from Garmin.
2. After creating the account, the application will show a list of different Garmin devices and asks the user the device he/she will use (in this case, the Vivosmart 4).
3. For this step, the smartband needs to be turned on. After selecting the respective device, the smartphone will try to connect via Bluetooth with the

7 https://github.com/topjohnwu/Magisk
8 https://odindownload.com/
9 https://topjohnwu.github.io/Magisk/install.html
10 https://www.droidwin.com/root-samsung-magisk-odin/

device. The process is done by typing a six-digit code shown on the screen of the smartband in a pop-up that will appear on the smartphone.

4. After successfully connecting both devices, the application will prompt the user to choose his/her profile image and to configure the privacy features, namely whether the user wants to share specific data with others or keep it private.
5. The application will ask for various data regarding the user: gender, height, weight, and birthday. The application will also prompt the user to define their sleeping time and in what arm they will use the device.
6. Lastly, the application will prompt the user to choose what interface the smartwatch should have and to accept the collection of the user data by Garmin.

In this work, the application was used with the Garmin Vivosmart 4. As we shall see later on, this smartband contains a variety of sensors, such as an optical heart rate monitor, a barometric altimeter, accelerometers, an ambient light sensor, and a Peripheral Capillary Oxygen Saturation (SpO_2) sensor. It connects to the smartphone via BLE and is compatible with Garmin proprietary interoperability ANT+ equipment protocol (Bang et al., 2022).

After everything is set up, the user is presented with the main dashboard of the Garmin Connect application. This dashboard is the central area of the application. Here the user sees the data collected by the Vivosmart 4 and the performed workouts. To update the dashboard, the user needs to synchronize with the smartband. Usually, this process is automatic, but the user can also do it by selecting the synchronize button. It is important to note that the application **only works with Internet connection**. If the user is offline, the application does not update.

For testing purposes and to collect as much data as possible, the Garmin Vivosmart 4 was used during various months. We used the following features of the application and smartband:

- Realizing workouts (both indoors and outdoors)
- Monitored Sleeping
- Changing body weight (defined by the user in the application)
- Adding Hydration Level (defined by the user in the application)
- Creating Running Routes (defined by the user in the application)
- Creating Exercises (defined by the user in the application)

Table 11: Most relevant functionalities of the Vivosmart 4.

FEATURE	DESCRIPTION
Heart rate	Measure heart rate.
Max. Oxygen Consumption	Measure the Maximal Oxygen Consumption also known as VO_2 using the heartbeat data.
Oximeter	Measure the oxygen saturation level in the blood, also known as SpO_2.
Steps	Count the number of steps.
Body Battery	Measure Body Battery (energy levels based on sleep and calories consumed).
Floors	Count number of floors ascended and descended.
Stress	Detect and measure stress levels.
Calories	Count calories burned.
Sleep	Analyze sleep along four main components: deep sleep, light sleep, awake time, and rapid eye movement (REM).
Workout	Allow selecting one of the four default workouts (walking, running, strength, yoga). Using Garmin Connect, it is possible to change or add new workouts.
Notification	Vivosmart 4 vibrates and displays a message for the smartphone notifications (e.g., calls, SMS, WhatsApp, *etc*).
Time	Time-related functions such as chronometer, countdown and timer.

- Adding Equipment (defined by the user in the application)

COLLECTION After data generation, the second phase of the static analysis is retrieving the information from the device.

The main areas in the Android Storage System are `internalstorage` and `externalstorage` [11] commonly called the private and public directories.

By default, the application-generated data is saved to the internal storage, which is private to the application. Other apps cannot access them or users unless they have root access. As the name implies, internal storage is a good place for application data that the user does not need to access directly, such as database files, application logs, and related data.

When a user uninstalls an application, saved files in the internal storage are removed. Because of this, internal storage should not be used to keep data persistently. For instance, when it comes to captured photos, storing these files on an external storage device is recommended.

Every Android device supports a shared "external storage" space for developers to save files. Files saved to the external storage are world-readable and can be modified by the user when they enable Universal Serial Bus (USB) mass storage to transfer files on a computer.

For a forensic practitioner, the goal is to extract the data stored on the internal and, if it exists, and on the external storage system. This process can be done using forensic tools like Cellebrite. However, here, we resorted to ADB. ADB is a command-line tool for communicating with an Android device connected to the computer via USB or wireless (since Android 11). ADB facilitates various device actions, such as installing and debugging apps. It provides access to a Unix shell that the user can use to run multiple commands on a device [12].

The process of extracting the data using ADB is the same for any application:

1. Access the device via ADB
2. Enter the Public or Private Directory (requires `sudo` privileges)
3. Find the applications folder
4. Archive and compress it and store it in external storage
5. Pull it from the device to the analyst's workstation

11 https://developer.android.com/training/data-storage
12 https://developer.android.com/studio/command-line/adb

To proceed, the following action involves utilizingADB to extract all data stored within the application and its associated file. We started by extracting the public folder. This data is stored in the location:

```
/storage/emulated/0/Android/data/
```

This folder holds subfolders, one per installed application on the device. Android follows the Java standard naming convention for its packages. This process is done by writing the package name in hierarchical order in the following way:

1. com (commercial application)
2. Name of the organization
3. Subdivision of the package (optional)
4. Name of the application

The package name for the Garmin Connect application is `com.Garmin.android.apps.connectmobile`. As stated earlier, data in a public folder requires no root privileges and is extracted through ADB.

After that, the next and most important part is extracting the application's private folder. This folder is stored in the path:

```
/data/data/
```

However, unlike the public directory, the user needs root access. Assuming we are dealing with a rooted device, we only need to execute the command `su` to enter privileged mode and access the private directory. After that, the process is the same, extracting the application package.

We also extracted the Android Application Package (APK) since some tools required this file to perform the analysis. The APK files are stored in the folder:

```
/data/app/
```

Since Android 8, the name of the app folders is named using a random string in base64. The goal of this feature is to enhance privacy and security. Using a random base64 naming scheme makes it more challenging for unauthorized users or malicious applications to identify and access sensitive app data (Developers, 2023). We needed to execute the following command to help find the folder connected to the app package to get the correct path.

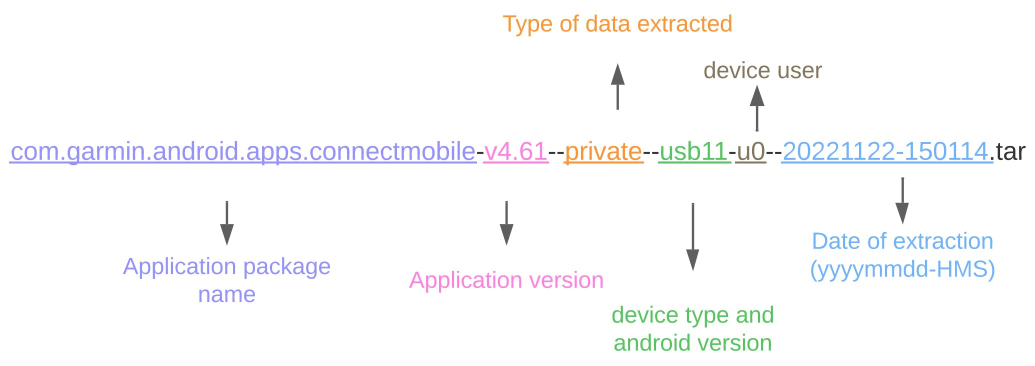

Type of data extracted
device user
com.garmin.android.apps.connectmobile-v4.61--private--usb11-u0--20221122-150114.tar
Application package name
Application version
device type and android version
Date of extraction (yyyymmdd-HMS)

Figure 4: Output file fields

```
ADB shell pm path com.Garmin.android.apps.connectmobile
```

The commands output the path to the location of the APK file of the application. After that, the user only needs to extract the file using ADB and the respective path.

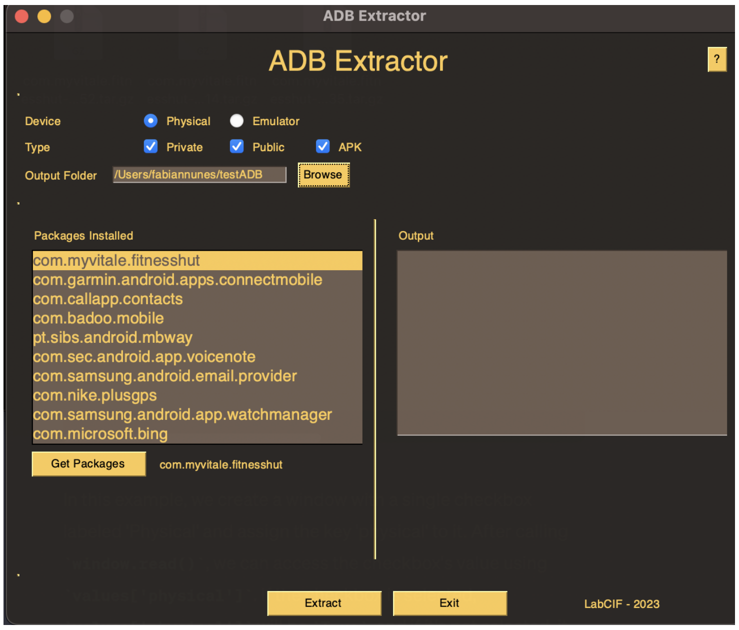
As we needed to extract all this data several times, this task could become quite time-consuming, so we created a script to automate this process. The script took inspiration from the past bash script created by Professor Miguel Frade [13], that extracted the private directory from a phone or emulator. The original script only worked on Linux systems failing to operate on macOSX systems. Thus, we decided to rewrite the script resorting to Python and be system agnostic to support the main OS, ie. Linux, Windows, and macOS. The script called **ADB-Extractor**, lets the user choose what he/she wants to extract the public directory, the private directory, or the application file from the device or an emulator.

To run the script, the user needs to type the command:

```
python3 acquisition.py -a <package name> -d [emulator | physical] -t [public |
↪  private | apk]
```

In the command, the user specifies the application's package name, if the acquisition comes from a physical device or an emulator, and what data he/she wants to pull. After running the command, the script will provide into the current directory, a compressed tar archive, e.g., `com.garmin.android.apps.connectmobile-v4.61 --private--usb11-u0--20221122-150114.tar`. Figure 4 illustrates the meaning of each field in the outputs name.

13 https://github.com/labcif/AndroidAcquisitionScript


ADB Extractor

ADB Extractor

Device [✓ Physical] Emulator

Type [✓ Private] [✓ Public] [✓ APK]

Output Folder /Users/fabiannunes/testADB Browse

Packages Installed

- com.myvitale.fitnesshut
- com.garmin.android.apps.connectmobile
- com.callapp.contacts
- com.badoo.mobile
- pt.sibs.android.mbway
- com.sec.android.app.voicenote
- com.samsung.android.email.provider
- com.nike.plusgps
- com.samsung.android.app.watchmanager
- com.microsoft.bing

Get Packages com.myvitale.fitnesshut

Output

Extract Exit LabCIF - 2023

Figure 5: Graphical Interface of ADB Extractor

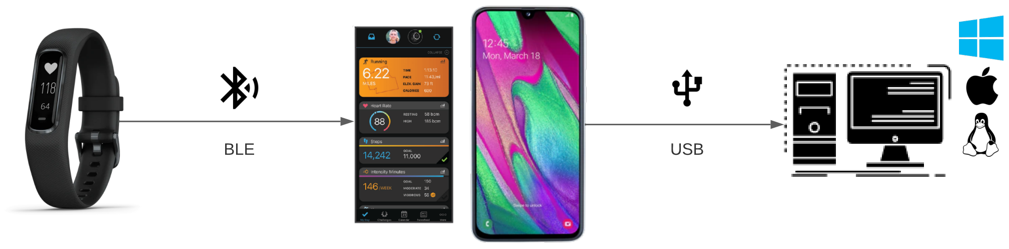


Figure 6: Acquisition Method used for this analysis

For example, to retrieve the private directory of the Garmin Connect application from a rooted phone, the user needs to run the command:

```
python3 acquisition.py com.garmin.android.apps.connectmobile -d -private
```

We also developed a graphical interface version that is similar to the **PySimpleGUI** library. In the graphical interface, the user can select the type of device, what to extract, and where it should be saved. Additionally, the user can select from a list what package he/she wants to extract based on the applications installed on the device connected via ADB. Figure 5 shows the created interface.

All script locations are detailed in Table 31 in Chapter 7. The acquisition process is exemplified in Figure 6.

In the end, we managed to extract the three main data sources from the Android device that are required for the static analysis, that is:

- Private Data
- Public Data
- Application Package

3.3 ARTIFACT ANALYSIS

After obtaining the data in the section 3.2, we analyzed data extracted from the device. As stated earlier, the analyzed data is split into three main components:

- The public directory
- The private directory
- The application file

These components have different purposes and will be used as input for the tools explained in the subsection 3.2.2 and analyzed manually.

3.3.1 *Public Directory*

The public directory, as explained in paragraph 3.2.3, is a directory where developers can store application data. This folder usually stores files that can persist even if the user uninstalls the application from the device.

After extracting the data to a safe location and unzipping it (it is advisable to perform this process within a POSIX environment due to the possibility of encountering file names incompatible with Windows. Failure to do so may result in potential data corruption.) the first step is to understand the file structure of the folder. The command `tree` will print all the subdirectories and files inside the directory where the command was executed. The public folder holds four directories and two files. In the Figure 7 is the output of the command `tree` with the following parameters:

```
tree -d -L 2
```

The content of the public folder is reduced. We only found two files: `map_cache.db` and `temp_file`. Using DB Browser to analyze `map_cache.db`, we found four tables, mainly with BLOB values and timestamps for expiration dates. The information stored did not have any forensic relevance. By searching for the name of the database

```
/sdcard/Android/data/com.garmin.android.apps.connectmobile/
├── cache/
│   └── diskcache/
├── files/
└── testdata/
```

Figure 7: Directories inside public folder

online, we discovered that this database stores cache tiles for the map functionality of the application to speed up its process during use. The `temp_file` was not human-readable and is likely used by Garmin Connect to save temporary data during the app's runtime. This file can be used for various purposes, such as caching, storing temporary data, or facilitating data exchange between components within the app.

The application did not store any relevant data in its public folder. There is no need to analyze this directory further or use other tools here.

3.3.2 *Private Directory*

The private directory is where the applications stores sensitive data like shared preferences, user credentials, and data stored in databases. Generally, the forensic analyst aims to search for artifacts in this directory since it is here that he/she can find potentially relevant information. Typically, data inside the private folder is secure since applications cannot access other applications' data, and the user cannot access this folder. However, since the used smartphone was rooted, we could access the data via ADB. That is one of the reasons some applications, as a security measure, detect root status and prevent the application from being used.

After extracting and saving the data securely, we made the same reconnaissance process we did for the public directory. That is executing the `tree` command to get an initial view of the data stored here. From this simple command, it was possible to understand that the private directory holds much more information than the public folder. The private folder has 51 subdirectories and contains 674 files. These numbers are based on our tests with this device. Other tests with other wearables can lead to a different number of files. However, many files do not hold forensic value, thus easing the analysis process. Figure 8 shows the private directory structure up to two levels.

```
/data/data/com.garmin.android.apps.connectmobile/
├── app_/
│   └── testdata/
├── app_OMT_ANALYTICS_DIR/
├── app_textures/
├── app_webview/
│   └── Default/
├── cache/
│   ├── WebView/
│   ├── com.google.android.gms.maps.volley/
│   ├── image_manager_disk_cache/
│   └── logs/
├── code_cache/
├── databases/
├── files/
│   ├── downloads/
│   ├── gcsync/
│   ├── logs/
│   ├── phenotype/
│   ├── upload/
│   └── uploads/
├── no_backup/
└── shared_prefs/
```

Figure 8: Directories inside private folder

APP FOLDERS We started by analyzing the folders in hierarchical order. We first noted that several folders had the `app_` prefix in their name. These folders are related to processes and actions done by Android when executing applications (Skulkin et al., 2018). Generally, these folders do not hold relevant data that can be considered a relevant artifact. Still, we analyzed the files inside of them.

The first folder shown is called `app_`. This folder only holds one file called `map_cache.key`. The file is not human-readable. Based on its name, it could be cached key values for the applications' Google Maps functionality. The second folder is called `app_OMT_ANALYTICS_DIR` and probably refers to the Google Analytics service. However, the folder is empty, like the folder called `app_textures`. In Android, applications can use this folder to store textures.

The last folder is called `app_webview`. This folder is used by Android applications that implement the WebView component. WebView is a system component for the Android operating system that enables Android apps to display web content directly inside an application. WebView lets applications display browser windows in an app instead of transporting the user to another browser [14]. The browser rendered by WebView is based on Chromium, so the data stored in this folder is very similar to one found in a Chromium browser installed on our computers. This folder has fewer

14 https://developer.android.com/reference/android/webkit/WebView

files than those found in browsers. The cause can be that the application does not use the WebView component for any functionality tested except for the signup, and to show the terms and conditions.

CACHE FOLDER The primary goal of the cache folder in Android is to provide a temporary storage location for data that can be recreated or retrieved easily. Caching improves performance and reduces network usage (Skulkin et al., 2018). Since the data that can be stored in the cache is so small, most of the files found are not human-readable and are only temporary files created very recently that hold not more than single digits of kilobytes of storage. Within this directory existed an image file. This file contained the cropped version of the profile picture of the user account.

`logs` is the only subfolder of the `cache` folder that held data. Specifically, the `logs` folder hosted the application's most recent `logcat` log. Logcat is a tool used in Android app development for logging information during the application's runtime. Developers use it to test their applications, debug features, and solve problems (Android Developers, 2023). Developers should not log sensitive data from the production version since log files are one way attackers steal data and find ways to attack the application. We found research from Cheng et al. (2021) that developed a proof of concept called **LogExtractor**. The tool automatically identifies and extracts digital evidence from log messages on an Android device. Given a log message, LogExtractor first determines whether it contains forensic artifact data (for example, GPS coordinates) and then further extracts the value of the data. Sadly Cheng et al. (2021) did not publish the tool.

We manually analyzed the log file inside the folder and searched for keywords that could indicate sensitive data used in the authentication process. The searched keywords were:

- auth
- expiration
- id
- key
- name
- password
- user
- token
- uid

No relevant data matching the provided keywords were found. The discovered data primarily pertained to the structure of the Android activities that the user accessed. Due to the limited timeframe of the analysis, covering only the last two days prior to the data acquisition, it is not possible to determine if the logcat logs contain any sensitive information. It should be noted that as the user did not engage in any login activities during that period, the likelihood of sensitive information being present is low.

The folder `com.google.android.gms.maps.volley`, contains cache data related to Google Maps since the applications use their API for the GPS functionality, while folder `WebView` stores cache from the WebView component, and lastly `image_manage_disk_cache` stores encoded bitmaps and images used by the application. The goal of this process is to load the pictures instantly without delay [15]. We tried to decode images but did not have success.

DATABASE FOLDER The database folder is where the `SQLite` databases of the application are stored. This folder contains various databases requiring extensive analysis, so we created a subsection 3.3.3 dedicated to this process.

In summary, the data retrieved is held in the databases where:

- Activity data
- Daily Data
- Device Values
- GPS Location
- Health Values
- Phone Notification
- Synchronization Logs
- User Profile Data

FILES FOLDER Typically, the directory `files` is linked to files generated by the application, whether through user interaction or not. This folder has various sub-folders and files, most of which were fragments of synchronization processes. Upon inspection, we observed that the files were not in a human-readable format. The Linux `file`[16] command identified the files as being in the Flexible and Interoperable Data Transfer (FIT) format.

`FIT` is a binary file format that stores health and fitness data such as workouts, heart rate, and GPS. It was developed by Garmin and is used by various fitness trackers and apps, including Garmin devices, Strava, and many others (Garmin Ltd., 2023). These files can contain data recorded by the device, meaning they might

15 https://developer.android.com/topic/performance/graphics/cache-bitmap
16 https://www.man7.org/linux/man-pages/man1/file.1.html

have forensic value. However, the only official way to decode a FIT file is using Garmins proprietary Software Development Kit (SDK) for developers. Luckily, the community developed open-source scripts to decode these files. One of these tools is a Python library called `fitdecode`[17], which converts a FIT file to JSON. However, the tool did not manage to decode these files since they are only fragments of a complete FIT file and are likely the result of caching processes.

We only found two human-readable files. The first one is a `JSON` file called `PersistedInstallation` that contains the `access_token` and `refresh_token`, as shown in listing 1. This file is related to Firebase, a real-time `noSQL` database developed and maintained by Google. The file contained the end user credentials such as the databases ID (Fid), the current Authentication token needed to make requests, the refresh token for when the authentication token expires, and two timestamps in seconds when the token was issued and its validity time. The authentication token is valid for seven days. The refresh token never expires (based on the firebase code public on Github [18]). This finding is not valuable from a forensic perspective. Still, it could be considered a security risk since an attacker could access this file and access the user's data accessible in the applications' Firebase database.

Listing 1: Content of `PersistedInstallation` file

```
{
  "Fid": "dOrIAtkqT66jHW4uNghIvs",
  "Status": 3,
  "AuthToken": "eyJ...(a total of 305 characters)",
  "RefreshToken": "3_A...(a total of 112 characters)",
  "TokenCreationEpochInSecs": 1677750057,
  "ExpiresInSecs": 604800
}
```

The second file, named `app.log` and stored in the subfolder `logs`, contains all the execution logs during the last day it was used. This file occupies a total of **3.43 Megabytes** over 22 582 lines. Therefore, we decided to search for specific keywords related to the authentication process, such as: `auth`, `token`, `secret`, `password`, and `id`. Using these keywords, we found interesting information about the application execution, even before analyzing it dynamically. The application logs the HTTP communications with Garmin's servers, as shown in the Listing 2.

In Listing 2, we can see a GET request to the URL `https://connectapi.garmin.com/mobile-gateway/snapshot/usageIndicators/v3`, using an authorization bearer token. Executing this request in Postman (this tool will be presented and

17 https://github.com/polyvertex/fitdecode
18 https://github.com/firebase/firebase-android-sdk/blob/master/firebase-installations/src/main/java/com/google/firebase/installations/local/PersistedInstallation.java

Listing 2: Content of app.log file

```
Mar-01;8:37:04.774PM [OkHttp https://connectapi.garmin.com/...] D/NetworkDI -
↪  --> GET
↪  https://connectapi.garmin.com/mobile-gateway/snapshot/usageIndicators/v3
↪  http/1.1
X-Garmin-Paired-App-Version: 7302
X-Garmin-Client-Platform: Android
User-Agent: GCM-Android-4.61
X-Garmin-User-Agent: com.garmin.android.apps.connectmobile/4.61; ;
↪  samsung/SM-A405FN/samsung; Android/30; Dalvik/2.1.0 (Linux; U; Android 11;
↪  SM-A405FN Build/RP1A.200720.012)
X-app-ver: 7302
X-lang: pt
Authorization: Bearer eyJ...(a total of 946 characters)
--> END GET
                                        ...
```

explained in Chapter 4 in the Table 19) returned a JSON response containing the features available on the device used in the Garmin Vivosmart 4. The Authorization token is a JSON Web Token or JWT used to authenticate the requests to the API and protect it against unauthorized access. Our testing found that this token remains valid for 24 hours before it expires. The token consists of 946 characters, less than the standard size for JWT tokens of 1 Kilobyte. This discovery will be precious in the dynamic analysis since the last logged Bearer token in this file could still be valid, meaning we can interact with the API without knowing the user's credentials. This fact is essential for this project because it means there is a chance of having access to all the user's data logged to Garmin Connect cloud servers. In the request, there is also information related to the client device that initiated the requests, such as the device model, the applications versions, and the current installed Android version. This information also holds forensic value since it could tell if this request came from the phone we are analyzing or from another that the user might be logged in.

SHARED_PREFS FOLDER The last folder to be analyzed is the `shared_prefs` folder. Shared Preferences stores XML files that hold small key-value data [19]. This method often stores small amounts of data the application needs. This removes the need to store data in a database, making the process simpler and quicker. Shared Preferences often store files holding data like usernames, email, and other unique data. This is why the files stored here can be an excellent place to find forensic artifacts.

The folder held 48 XML files in our experiments. Filtering all these contents can be time-consuming, so we first needed to make a superficial analysis to exclude files

19 https://developer.android.com/reference/android/content/SharedPreferences

that we did not consider to have forensic value, such as empty files and Files with no user data.

After this first filtering, we could exclude 32 (11 empty files and 21 files with no forensic value). The reaming 16 files that seemed to hold relevant content were:

- `com.facebook.AccessTokenManager.SharedPreferences`
- `com.facebook.sdk.appEventPreferences`
- `com.facebook.sdk.USER_SETTINGS`
- `com.google.android.gms.appid`
- `com.google.android.gms.measurement.prefs`
- `com.google.android.gms.signin`
- `com.google.firebase.crashlytics`
- `com.google.maps.api.android.lib6.drd.PREFERENCES\_FILE`
- `FirebaseAppHeartBeat`
- `gcm_user_preferences`
- `mobile.auth.sec`
- `samd_sec`
- `settings_preference`
- `ue3Preference`

The first files to be analyzed were related to the Facebook API. To use Facebook features in Garmin Connect, users must authenticate into his/her account and approve the connection. There was a token called `access-code` stored in the file `com.facebook.AccessTokenManager.SharedPreferences` file. Searching in the Facebook API documentation, we discovered that the application uses this access token when making requests to the API. Furthermore, several methods exist to explore the token to access Facebook user data [20]. As proof of concept, we resorted to the well-known Client URL (cURL) HTTPS client to interact with the API. Almost all resources required a `user id` for proper access. A `user id` is trivial to discover as shown in the following Listing:

```
curl --location --request GET
↪  'https://graph.facebook.com/me?access_token=EA...(a total of 250 characters)'
```

20 https://developers.facebook.com/docs/graph-api/overview

The cURL execution command returns a JSON response with the user name and the user id from Facebook. With the user id and the access code, we could access all data Facebook shares with Garmin Connect. The API offers a Uniform Resource Locator (URL) that shows all the information we can access, which is shown in the cURL below:

```
curl --location --request GET
↪  'https://graph.facebook.com/2350304655138824?metadata=1&access_token=EA...(a
↪  total of 250 characters)'
```

Through this method, one can list the following user data:

- User ID
- User Name
- Profile Picture
- Number of Friends
- Garmin App ID
- App permissions

We also tried to obtain data related to the application resorting to the `appid`, yet we were unsuccessful.

The `com.google.gms` files, Google Mobile Services (GMS) is a collection of applications and `APIs` developed by Google for manufacturers of Android devices [21]. Those files provide information to the user, such as the Garmin Connects account id. However, without access to the Google Cloud Platform account from Garmin Connect, it is challenging to know what `APIs` it uses or test the `APIs`. In the file `com.google.android.gms.signin` it was possible to discover:

- User ID
- User Name
- Profile Picture

The shared preferences folder also holds files related to the Firebase services. These files contained data related to the Firebase services used by Garmin Connect, such as the database ID. However, the data found in these files had no forensic use.

21 https://www.android.com/gms/

The file that contained the most user-related data was the file `gcm_user_prefereces`. It stores a large amount of data related to the user account logged in to the application, manualy:

- Birth Date
- Country
- Email
- Gender
- Google Access Token
- Height
- Language
- Location
- Profile Picture
- Sleep hours
- Timezone
- User Hand
- User Id
- User Level
- User Points
- User wrist
- Weight
- Whether the user is currently pregnant

The file contains a considerable number of user artifacts. However, we found all these artifacts during the database analysis. It can be useful if we do not have access to it.

The file `mobile.auth.sec`, as the name suggests, could be used to store authentication-related information of the mobile application. However, as shown in Listing 3, all data in this file is encrypted. The application most likely uses the AndroidX Security library (Jetpack Security). This library encrypts the shared preferences files using the algorithm Advanced Encryption Standard (AES) in case it is confidential data. Without the encryption key, we cannot decrypt the values stored here. This also means they are securely protected against unauthorized access. Another file with similar results was `samd_sec`, where we found five base64 encoded strings with encrypted content.

Listing 3: mobile.auth.sec

```xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
    <map>
        <string name="ASmhk4/+XysQyJClRN0SDXvVB6sSDlpWhx4CkfsL1tyuWmsxpeA0xkakSr⌋
        ↪  Jp">ATg1CHNWTHy/kOFH5yYAC8hA/srD3wSfRst5B6PnLRgXGln1puc=</string>
        <string name="__androidx_security_crypto_encrypted_prefs_key_keyset__">1⌋
        ↪  2...(a total of 3078
        ↪  characters)</string>
        <string name="__androidx_security_crypto_encrypted_prefs_value_keyset__"⌋
        ↪  >12...(a total of 912
        ↪  characters)</string>
        <string name="ASmhk48sl4yLajgCPBypflbnF/mn3WDLMEyNb0gs7ba9Vq0Z">AT...(a
        ↪  total of 52 characters)</string>
        <string
        ↪  name="ASmhk4+/QMF5JAkiwhMyAgZwwgzAjy7re49e6ELfR8Jt5DBBZg==">AT...(a
        ↪  total of 60 characters)>/string>
    </map>
```

To conclude this analysis, we identified two more files that contained data but did not have forensic value, the `settings_preference`, which includes data similar to the maps file, and the `ue3Preference`, which contains various IDs. Yet, we could not establish its connection and use case for the Garmin Connect application.

3.3.3 *Database Analysis*

The primary and time-intensive phase of static analysis in an application involves examining its databases, typically located in the `databases` folder within the application's private directory. This step is crucial for analysts as it often uncovers most artifacts. In our case, the folder contains 17 `SQLite3` databases.

The large number of databases could mean that some are legacy while others are specific to the features used by the application.

The process to analyze the database was the following:

1. Open the database with DB Browser and identify valuable tables
2. Describe the database by analyzing the tables
3. Identify potential artifacts
4. Try to recover deleted data with the Python script `bring2lite`
5. Draw database diagram with schemacrawler and dbdiagram if forensicaly relevant

SQLITE FILE TYPES Before analyzing each database file, it is essential to note that SQLite stores the complete database in a single disk file and uses many temporary files while processing a database. We found three different types of temporary files in the database folder: **Rollback journals**, Write-Ahead-Log (WAL) files, and Share Memory File (SMF).

A rollback journal is a temporary file that implements SQLite's atomic commit and rollback capabilities. The suffix `-journal` identifies the rollback journal. The goal of the rollback journal is to prevent partial or total corruption of the database in case of an unfinished transaction or power loss. The rollback journal is usually created and destroyed at the start and end of a transaction, respectively.

A WAL file is used in place of a rollback journal when SQLite is operating in WAL mode. The goal of the WAL file is the same as the rollback journal. The file has the suffix `-wal` appended to the name of the database (Gaffney et al., 2022). Both rollback journal WAL files are mechanisms used in database systems to ensure data consistency and durability. However, they serve different purposes and have different characteristics. The rollback journal focuses on undoing changes made during a transaction, while the WAL file ensures durability and recovery of the database in case of failures.

The SMF, when it exists, is identified by the suffix `-shm`. SMF only exists while running in WAL mode and contains no persistent content. The only purpose of the file is to provide a block of shared memory for use by multiple processes, all accessing the same database in WAL mode. It is essential to talk about the temporary files created by SQLite because, as we said before, these files are atomically executed when the user opens the database in DB Browser. This can lead to data being deleted from the database and potential artifact loss.

Three methods exist to open the database file without executing the journal or WAL files. We rank them from the most viable to the least to prevent data alteration:

1. Use a forensic tool built for dealing with databases such as Autopsy
2. Copy the database file to a different folder and open it with DB Browser
3. Change the database file name and open it with DB Browser

The first method is the best yet. The forensic tools that have these capabilities are typically commercial solutions. The second method is the one that will be used here, first opening the database in a different location of the WAL file and then with the WAL file to see the differences. The last method is commonly used and works, yet, analysts have shown that this method can lead to confusion and loss of artifacts (Angie, 2016).

DATABASE STRUCTURE To better understand what each database does, we compiled a list of all databases in a large table. Table 12 presents the databases and briefly describes them. We also provide, for each database, the number of tables that hold records, as we observed that many tables were empty.

`APPLICATIONS-DATABASE` The first database analyzed was `applications-database`. The tables do not hold any forensic value, as the database stores Android metadata [22].

22 https://developer.android.com/reference/android/arch/persistence/room/Room

Table 12: Brief description of Garmin Connect databases. The column Tables displays the total of tables and, within parenthesis, the number of tables with data.

DATABASE	# tables		DESCRIPTION
`applications-database`	3	(3)	Related to the version of the application
`AppNotification`	3	(3)	Internal Notification of the application
`cache-database`	27	(14)	Data cached from the app's features
`Campaign_Database`	3	(2)	Possible storage for Garmin Campaigns
`com.google.android.` `datatransport.events`	7	(2)	Android internal communication process (Google Play Services library)
`connect`	4	(3)	mac address of the smartband
`garminpaycore`	4	(2)	Data regarding the Garmin Pay feature (not available for Vivosmart 4)
`gcm_cache.db`	12	(6)	Cache Data originated from the data received from the API
`gcm_onboarding_item`	5	(4)	Devices associated with account and features activated
`gcm_swings`	3	(2)	Database related to Golf features present in some devices (not available for Vivosmart 4)
`gcm_user_presistence`	7	(3)	Cached information for certain features of the application (onboarding and badges)
`google_app_measure` `↪ment_local.db`	2	(1)	Database related to Firebase Analytics
`livetrack-database`	6	(2)	Data regarding livetrack feature (not available for Vivosmart 4).
`news_feed_database`	3	(2)	Database related to the user feed.
`notification-database`	3	(3)	Database with the recent phone notifications sent to the smartband
`sync_cache.db`	3	(3)	Database with the synchronization process between smartband and application
`ue3.db`	3	(1)	Database possibly related to internal events

Table 13: `AppNotifications` Table

TABLE	# COLUMNS
`android_metadata`	1
`notifications`	5
`sqlite_sequence`	2

APPNOTIFICATIONS The database had three tables. Table 13 shows the database's tables. This database does not have forensic value. One of the tables is called `notifications`, but from analyzing the table, it is unrelated to sending notifications. One of the columns of this table is called `date` and stores the data in a UNIX Timestamp referring to the Coordinated Universal Time (UTC) time. After converting it to the current time, it indicates when the application was first launched on the phone. When launching, the application probably uses the database to start and store specific services.

CACHE-DATABASE We considered the `"cache-database"` database one of the most crucial in the whole application. Compared to the other databases, it is vast and holds 27 tables, with the table `activity-details` having as many as 120 columns.

This database holds the most meaningful digital forensic data. However, from the 27 tables, only 14 of those had data based on our setup.

We suspect that the absence of data in some tables is due to the lack of features from the Vivosmart 4 smartband that are present on other Garmin products (*e. g.*, blood pressure readings). These tables are listed in Table 14 and classified regarding their digital forensics value.

In the database, the four activity tables are related to the workout activities stored by Garmin Connect. They are connected by a foreign key which is the activity ID. To facilitate the analysis of this database, we used `schemacrawler` and `DBDiagram.io` to create a database diagram overview, shown in Figure 9.

First, we generated the database diagram with `schemacrawler` and the `SQLite` database. can detect the connections among the tables in a database and produce a corresponding diagram. The user can customize the level of precision in the diagram. This is done with the following four commands:

Table 14: Non-empty tables of the `cache-database` database

TABLE	FORENSIC VALUE	DESCRIPTION
`acclimation_pulse_ox_details`	fair	Recored SpO_2 data
`activity_chart_data`	high	Values for activities' charts
`activity_details`	high	Details of activities
`activity_polyline`	high	GPS coordinates from activities
`activity_summaries`	high	Activity Details in JSON format
`heartrate_zones`	low	Users Heart Rate Zone Values
`intensity_minutes`	low	Total Minutes of intense exercise
`response_cache`	low	Response Cache from server
`sleep_detail`	high	Stores Sleep Information
`user_daily_summary`	high	Daily user summaries
`weight`	fair	Weight Data

```
schemacrawler --shell
connect --server=sqlite --database=cache-database
load --info-level=maximum --weak-associations=true --infer-extension-tables
execute --command=schema --output-file=./db.png
```

After that, we improved the initial diagram using `DBDiagram.io`. Since the complete diagram is too big to present in the paper, we created a repository with the complete diagram and code to generate it in our repository presented in Table 31.

Each record in these tables represents a workout activity performed by the user. The two tables that hold more information are `activity_details` and `activity_summaries`, which contain the various details of an activity done by the user (calories burned, steps, heart rate, distance, *etc*). These two tables hold the same data. The only difference is that `activity_details` saves the data in separate columns (the table has a total of 120 columns), and `activity_summaries` saves all data related to the activity in a JSON object stored in a single column. We hypothesize the table `activity_summaries` stores the activity in the format sent to the server when the user uploads an activity. The table `activity_chart_data` is related to the charts generated for the activity. To create the chart, contains two columns that store an array of **X** and **Y** values. During our data gathering, we only managed to generate records related to heart rate charts. The table `activity_polyline` is associated with outdoor activities with GPS tracking. The table stores the starting and ending

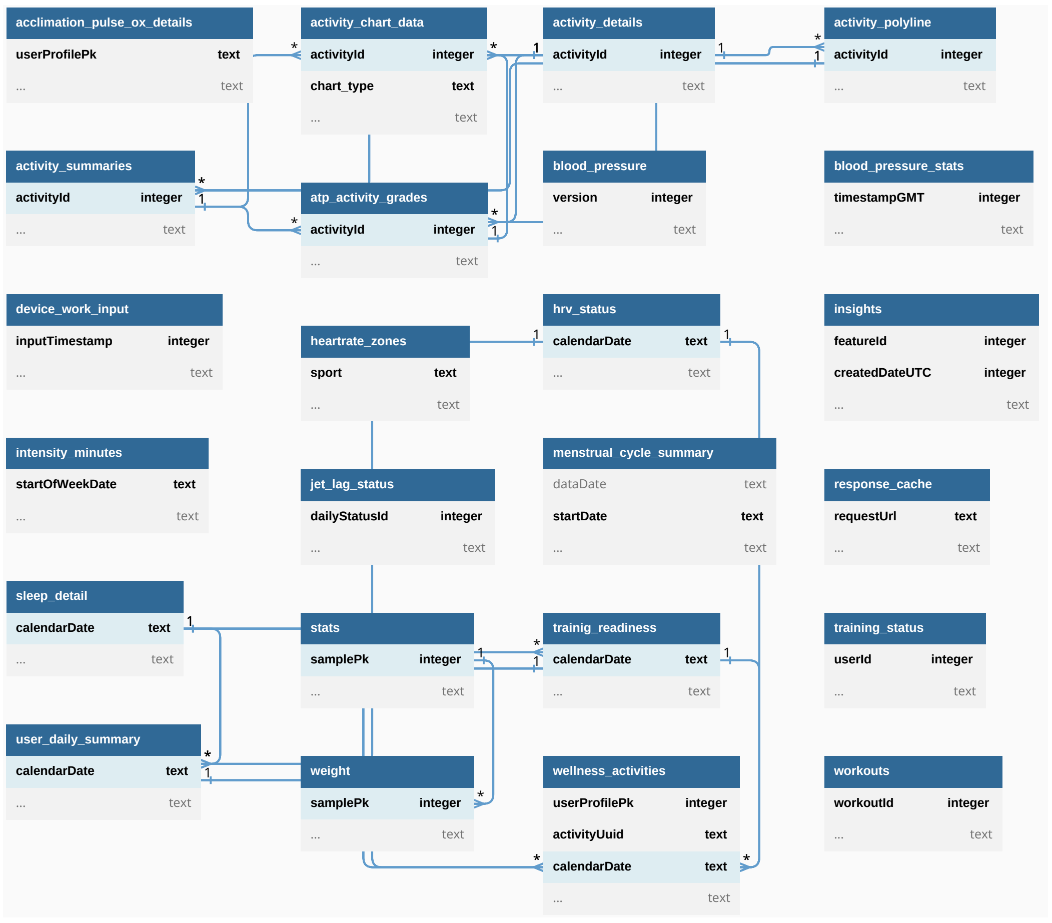


Figure 9: Simplified diagram of the cache-database

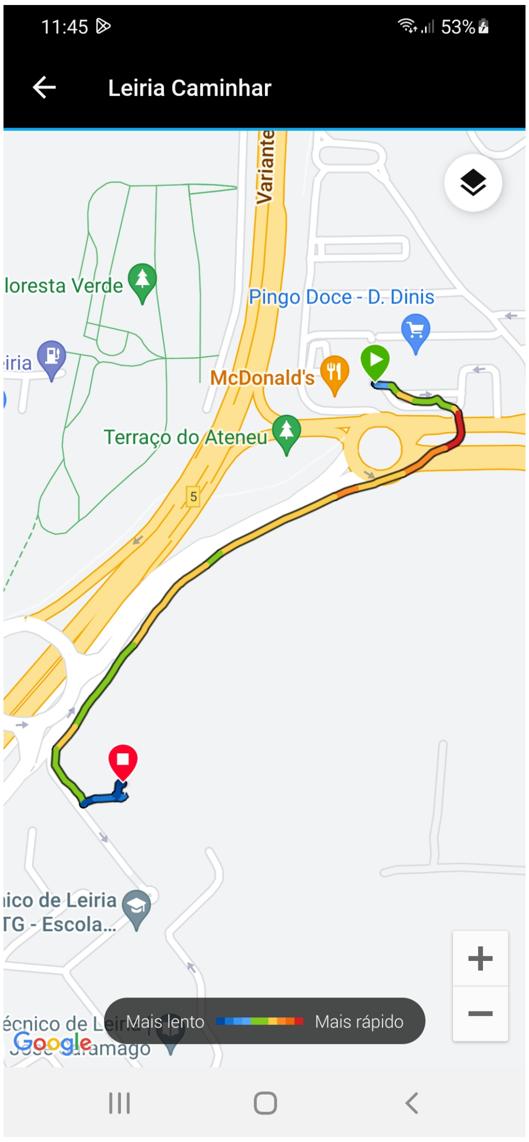
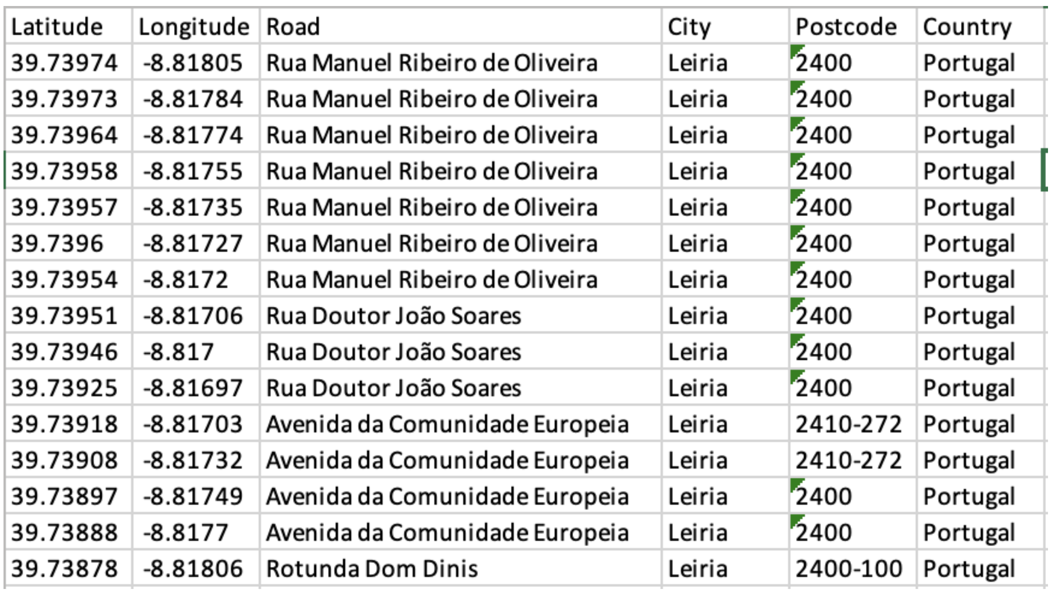


Figure 10: GPS Route of a walking activity

coordinates and a large string of characters called **polyline**. A polyline is a string of characters that encodes a series of coordinates. The polyline is encoded using the Google Maps API and is usually used to draw a route in Google Maps [23] as shown in the Figure 10.

The coordinates hold significant forensic value because they can pinpoint the user's location at a given time. According to Google's documentation, decoding a polyline back to group coordinates is possible. Using the Python library **polyline**[24], we created a Python script called `Polyline2GPS` that decodes Google's polylines back

23 https://developers.google.com/maps/documentation/javascript/examples/polyline-simple
24 https://pypi.org/project/polyline/



Latitude	Longitude	Road	City	Postcode	Country
39.73974	-8.81805	Rua Manuel Ribeiro de Oliveira	Leiria	2400	Portugal
39.73973	-8.81784	Rua Manuel Ribeiro de Oliveira	Leiria	2400	Portugal
39.73964	-8.81774	Rua Manuel Ribeiro de Oliveira	Leiria	2400	Portugal
39.73958	-8.81755	Rua Manuel Ribeiro de Oliveira	Leiria	2400	Portugal
39.73957	-8.81735	Rua Manuel Ribeiro de Oliveira	Leiria	2400	Portugal
39.7396	-8.81727	Rua Manuel Ribeiro de Oliveira	Leiria	2400	Portugal
39.73954	-8.8172	Rua Manuel Ribeiro de Oliveira	Leiria	2400	Portugal
39.73951	-8.81706	Rua Doutor João Soares	Leiria	2400	Portugal
39.73946	-8.817	Rua Doutor João Soares	Leiria	2400	Portugal
39.73925	-8.81697	Rua Doutor João Soares	Leiria	2400	Portugal
39.73918	-8.81703	Avenida da Comunidade Europeia	Leiria	2410-272	Portugal
39.73908	-8.81732	Avenida da Comunidade Europeia	Leiria	2410-272	Portugal
39.73897	-8.81749	Avenida da Comunidade Europeia	Leiria	2400	Portugal
39.73888	-8.8177	Avenida da Comunidade Europeia	Leiria	2400	Portugal
39.73878	-8.81806	Rotunda Dom Dinis	Leiria	2400-100	Portugal

Figure 11: Excert of the Excel generated by `Polyline2GPS`

to coordinates and saves them in a `XLSX` file. This file will contain the coordinates of the activity. Additionally, with the use of the library **geopy**[25], our script will also add information to the coordinates such as the respective **road**, **city**, **postcode** and **country**. Figure 11 shows an extract of the Excel generated by the script containing the data obtained from **geopy**.

This file aims to aid the analysis of GPS coordinates and filter possible locations of interest. After that, we use these coordinates to create a file with the route done by the user. The user can choose to export this file in `HTML` or Google Earth format (`KML`). This script was part of the modules developed for ALEAPP. However, since we could not find any script that did those features, we decided it could prove helpful for other use cases, so we created a standalone version called **Polyline2GPS**.

The table `sleep_detail` contains users' sleep data. It stores a timestamp when it starts the sleep mode. When it stops, the table also stores the duration in seconds of the recorded sleep phases (light sleep, deep sleep, REM sleep, and awake time). In addition, the Vivosmart 4 actively reads the user's SpO_2 during sleep and stores the lowest, highest, and average SpO_2 readings. The values of the table have forensic value since it can tell the analyst the timespan the person was asleep and correlate it with other events.

The table `user_daily_summary` stores general data such as calories burned, steps, stress, heart rate, SpO_2 *etc.* The table has 70 columns and stores the user's daily data (one record per day). This table is valuable from a forensic standpoint since it

25 https://pypi.org/project/geopy/

is possible to understand the user's day, such as the maximum heart rate and the number of steps.

The general problem with the `cache-database` is that the database only stores data temporarily. We used the smartband for various months, and only the most recent data was saved in the database. After synchronizing with the device, the application sends the data to Garminin's cloud servers, where the data is then stored. This process is part of the Garmin Connect API developed to share data with partner apps. That means that the application only uses the data in this database for caching and to reduce load times. The primary way to access the application's data is by retrieving it from the cloud, which is why it requires a continuous internet connection. While this slightly diminishes its forensic value, the fact that the application stores recent data in its cache still makes its information valuable.

By default, `SQLite` does not delete records. Instead, it marks them as unused until other data overwrites them. To retrieve any possible deleted records from the database, we utilized the open-source script **bring2lite** [26]. However, we could not retrieve any data. We later verified in the database properties that it has the flag `PRAGMA schema.auto_vacuum` set to `FULL`. With this option, SQLite moves the free list pages to the end of the database file. Then the file is truncated to remove them at every transaction commit, eliminating the possibility of retrieving deleted data (SQLite Community, 2023).

CAMPAIGN_DATABASE This database was empty. Based on the keyword **"Campaign"** it can mean the publicity campaigns or events that occurred in the application.

COM.GOOGLE.ANDROID.DATATRANSPORT.EVENTS This database is empty, yet by the file's name, it is possible to see that it is related to an internal Android service the application uses. By searching for the database name, we found a maven repository that points to the Android Transport API [27]. The Google documentation makes it possible to say that this database is related to data stored temporarily between network communications[28]. Yet, since it is empty, it does not hold value for this analysis.

26 https://github.com/bring2lite/bring2lite
27 https://mvnrepository.com/artifact/com.google.android.datatransport
28 https://developers.google.com/android/reference/com/google/android/gms/fido/common/Transport

Table 15: Non-empty tables of the `gcm_cache.db` database

TABLE	FORENSIC VALUE	DESCRIPTION
`device_permission`	low	Permissions of the device
`devices`	fair	Information related to the connected device
`json`	high	Device captured Data (`JSON`)
`json_activities`	high	Stored activities (`JSON`)

CONNECT The `Connect` database looked promising at first glance since it has the same name as the application, meaning it could store data related to the Garmin Connect application. That is true, yet it holds minimal information. The database holds four tables. However, only one held data: `device-capabilties`. This table stores the `userid` and `macaddress` from the paired smartband, which can prove valuable in identifying paired devices.

GARMINPAYCORE `Garminpaycore` is a database that could prove interesting for future analysis, but currently, it does not hold artifacts. The database is related to the Garmin Pay function presented in some of the Garmin Watches [29]. Garmin Pay is a service that lets users associate credit cards or bank accounts with Garmin Pay to use their smartwatch to perform contactless payments. The service is similar to Apple Pay and Google Pay.

GCM_CACHE The database `gcm_cache.db`, just like the `cache-database`, holds temporary data. It has various tables related to user data (activity information and daily statistics) and the associated devices. From the 12 tables inside the database, only 4 had any data after our tests, as presented in Table 15.

The table `device_permission` stores the permissions the user gives on the application. From what we gathered, different permissions on the application are identified by an id. The table stores whether the user gave permission (0 or 1). The table `devices` stores vast information from the Vivosmart 4 band (name, MAC address, id, capabilities). The `devices` contains valuable data, however with little forensic value.

29 https://www.garmin.com/pt-PT/garmin-pay/

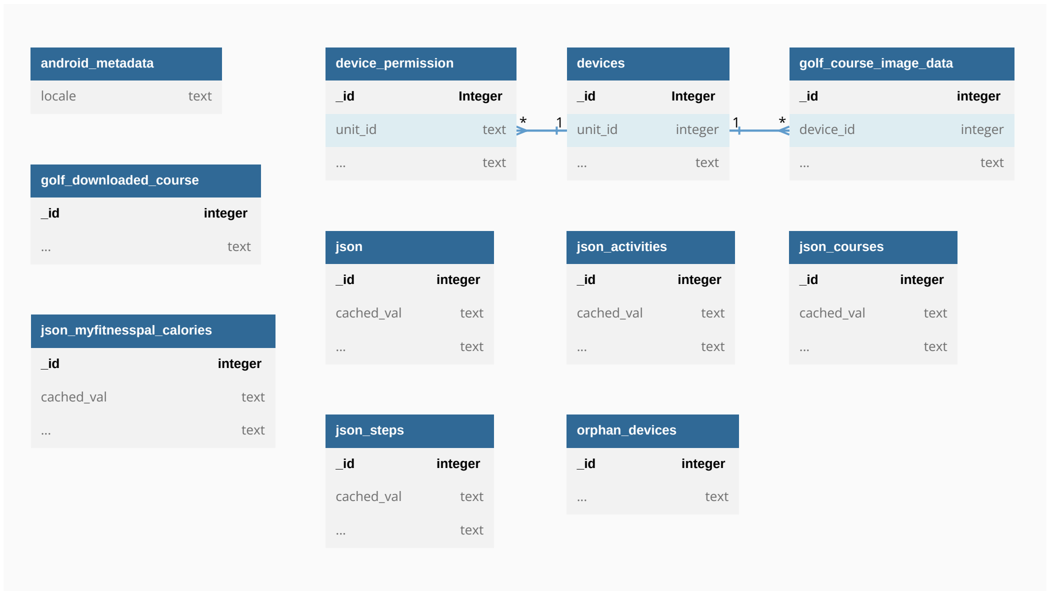


Figure 12: Simplified diagram of the GCM_Cache

The `gcm_cache.db` holds various tables with the prefix **json**. Only two of them had data, a table called `json` and another called `json_activities`. The tables store cached values in JSON format. The data present here is the same as in the `cache-database`. We suspect this database holds a cache from the data the application retrieves from the cloud via requests to the API. The `gcm_cache.db` and the `gcm_cache.db` hold the same data in different formats. The `cache-database` generally holds more information and is easier to read. Yet, we found records in the `gcm_cache.db` that were not present in the `cache-database`, meaning that both have forensic value and should be equally studied.

Just as before, we generated the database diagram in `schemacrawler` and refined it with `DBDiagram.io`. Figure 12 shows a cropped part from the resulting diagram.

`GCM_ONBOARDING_ITEM` This database stores information related to the first time the user opened the application and when he/she synchronized the device for the first time. It stores the available features and if they are enabled or not. If they are not, the application will remind the user about them the next time he opens them. This database does not hold forensic value for this analysis except for the possibility of knowing the date/time of the first usage.

`GCM_SWINGS` This database is related to the Garmin Golf application. It has only one table, which stores data related to golf equipment. Since the golf application was not tested, this database remained empty.

GOOGLE_APP_MEASURMENT_LOCAL This database is similar to the `com.google.android.datatransport` database, as it is not related to Garmin Connect but to a third-party technology used in the application. In this case, this database is related to Firebase Analytics. This tool provides essential performance and user experience metrics, such as [30]:

- User demographics
- Engagement and retention
- Crash rate
- Conversion events
- Deep-link performance

LIVETRACK-DATABASE This database does not contain data. It stores data related to a feature in some of the Garmin smartwatches called **Livetrack**. This feature lets users share their current path in real time with other users. Nevertheless, the absence of this feature in the Garmin Vivosmart 4 indicates that it is not available within the Garmin Connect application. [31].

NEWS_FEED_DATABASE We believe this database is related to the social feature of the Garmin Connect application, where the activities are similar to social media posts shared in the user feed. Friends can like and comment on it. We tried to use these features in our activities, yet the table remained empty, so it is difficult to tell if it would hold any forensic value.

NOTIFICATION-DATABASE The database `notifications-database`, as the name implies, is used to store the phone's notifications transmitted to the smartband. Since the notification appears in cleartext, it can be a good artifact, for example, possible incriminating text messages or call logs. Unfortunately, this database only stores recent notifications and frequently deletes old notifications. We have verified that this database, similar to `cache-database`, has the flag `PRAGMA schema.auto_vacuum` set to `FULL`, thereby rendering the recovery of deleted records impossible.

30 https://firebase.google.com/docs/analytics
31 https://www.garmin.com/en-US/blog/fitness/use-garmin-livetrack-track-activities-real-time/

SYNC_CACHE The database `sync_cache` stores the synchronization process of the smartband and the smartphone. This database contains forensic value since it gives us some information, such as what smartband is synchronized with the smartphone (the database stores the unit ID of the device in each record). It also tells us that the smartphone connected the device via `BLE` at that moment – each record contains a `UTC` timestamp.

UE3 This database only has one empty table. By the name of the database, it is impossible to know what the goal of this database is or was (most probably). We omitted it from the document, as it looks like a legacy database with zero value. Thus, we do not provide an analysis of the table.

3.4 MOBSF ANALYSIS

After analyzing the data stored by Garmin Connect in the private and public directories, the only step left is to analyze the APK file. This file does not contain data related to the user. Discovering the privacy and security aspects of the application can be crucial before starting the dynamic analysis. This process is done in forensic research and by developers to confirm that they follow the necessary regulations, such as General Data Protection Regulation (GDPR) and Health Insurance Portability and Accountability Act (HIPAA), to name just a few. We will use the all-in-one scanning tool MobSF.

The process of installing and configuring MobSF can be lengthy for that reason we created the appendix A where we explain this process in detail.

3.4.1 *Background study*

MobSF is an automated, all-in-one mobile application pen-testing, malware analysis, and security assessment framework capable of performing static and dynamic analysis[32].

Other tools work as alternatives or competitors to MobSF, such as **Metasploit** or **Veracode**. The reason for choosing MobSF was:

- Easy-to-use scanner for static analysis

32 https://github.com/MobSF/Mobile-Security-Framework-MobSF

- Creation of a web report and an associated interface to navigate the data
- Free tool with constant community support and improvements
- Customizable if needed

MobSF is generally used in security assessments and research. The authors analyzed various papers that used MobSF. Sachdeva et al. (2018) used MobSF with machine learning techniques to classify mobile applications based on their files. One of the core features of MobSF is its malware analysis module, which will not be used in this research. Research by Shahriar et al. (2019) evaluated four security tools: Flowdroid, CuckoDroid, MobSF, and Droidbox. However, the study did not go into much detail and was focused primarily on malware analysis.

Another research used MobSF to analyze 204 health-related Android applications and, using the OWASP Top 10, evaluated the number of security flaws presented in the applications. The study found that 43.62% of the applications had at least one security flaw and 26.47% had at least two security flaws (Lamalva and Schmeelk, 2020). Their study also revealed that the flaws are mainly centered in the following categories:

- Improper Platform Usage
- Insecure Data Storage
- Insecure Cryptography
- Client Code Quality
- Reverse Engineering

Lastly, Barros et al. (2022) used MobSF during their analysis of the Bumble application to discover various privacy functionalities.

3.4.2 *Analysis*

After the process is terminated, MobSF generates a PDF report with all the information collected. The user can browse all the data through the web interface or resort to the PDF report.

The generated report is quite large, with 106 pages. Since the web dashboard offers a better user experience, it was our primary analysis method.

MobSF generates two different areas related to static analysis. It creates a view called **AppSec Scoreboard** that contains an overview of the static analysis showing a list of the findings and a view called **Static Analysis** where the user can interact with everything done during this analysis.

In the report MobSF divides the content through different topics, we will present the topics that contained the most significant findings, they were:

- Network Security
- Certificate Analysis
- Maniefest Analysis
- Code Analysis
- Reconnaissance

APPSEC SCOREBOARD We started by analyzing the scoreboard. Here we are presented with various statistics. Based on the vulnerabilities found in total, MobSF gives a security score ranging from 0 to 100 and a risk rating from A to F (Best to Worst). In our case, Garmin Connect acquired a security score below the average of 45 and a **B** risk rating that is considered a medium risk, as shown in Figure 13. These values are concerning considering the large user base that Garmin Connect has. Yet, it is unfair to judge the application solely on the score given by an open-source tool that solely performs static analysis. Therefore, we need to continue the analysis to reach a particular conclusion.

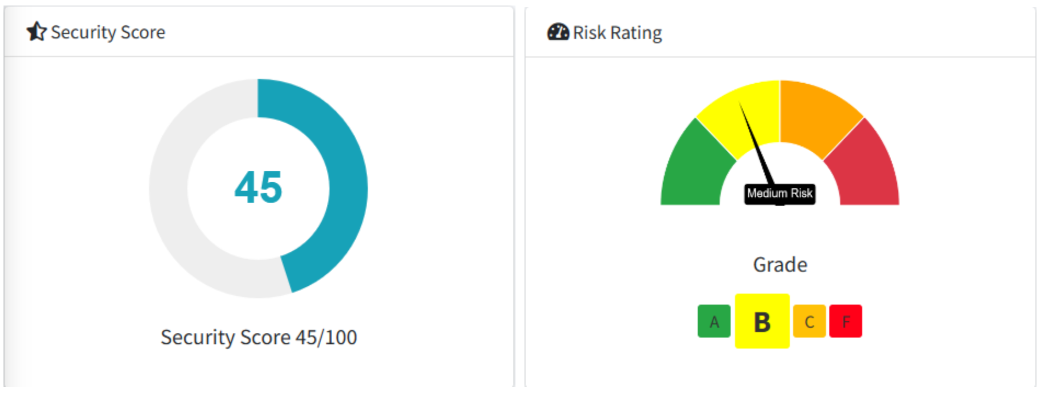
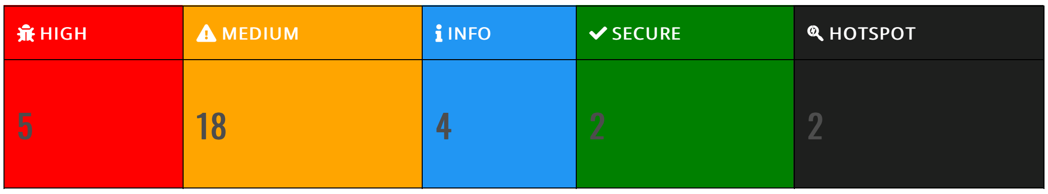


Figure 13: Mobsf Security Score (Left) and Risk Rating(Right)

The scoreboard also shows the findings discovered by MobSF. The scanner found 31 results in our case and split them into five categories **High, Medium, Info, Secure, and Hotspot**. The focus is generally on High and Medium findings since these represent vulnerabilities, while the info category only contains possible warnings. Secure is related to correct security implementations detected and hotspot to files of interest. Figure 14 shows all findings collected by MobSF.



HIGH	MEDIUM	INFO	SECURE	HOTSPOT
5	18	4	2	2

Figure 14: MobSF Findings

The scoreboard also shows the number of trackers the application has. A tracker refers to a component or functionality within a mobile application that collects and transmits user data to third-party entities without the user's explicit consent or knowledge. Trackers are often used for various purposes such as analytics, advertising, profiling, and user behavior tracking (Razaghpanah et al., 2018). In this case, MobSF detected four trackers. In the following paragraphs, we will use the static analysis dashboard and analyze the findings to see if there is valuable data to extract.

PERMISSIONS MobSF identifies all permissions required by the application in its `AndroidManifest` file. The tool identified 49 permissions and classified 17 as critical or dangerous, as they can be exploited by a malicious application. Table 16 displays the permissions classified as dangerous.

Cross-referencing these permissions with the ones told by Garmin on the Play Store page, we can confirm that the application did not ask for permission not presented to the user before installation.

NETWORK SECURITY This topic is related to the network settings of the application. Here, the scanner identified a finding with high severity described as *Domain config is insecurely configured to permit clear text traffic to these domains in scope*. This description means that the developers defined in the XML configuration file the data to be transmitted in plain Hypertext Transfer Protocol (HTTP) for the domains shown in Listing 4.

Sending data through HTTP on the local host is a common practice for development purposes. However, it should never be used in production since it can send sensitive data through clear text that an attacker can collect during the communication. It is also interesting to note how Garmin Connect sends the data in clear text to the application Strava since this could potentially be considered a security and privacy flaw when dealing with personal data.

Table 16: The application's permissions classified as dangerous

PERMISSION	DESCRIPTION
android.permission.WRITE_EXTERNAL_STORAGE	Allows the application to, write to external storage.
android.permission.READ_EXTERNAL_STORAGE	Allows the application to, read from external storage.
android.permission.ACCESS_FINE_LOCATION	Allows access to a precise location.
android.permission.ACCESS_COARSE_LOCATION	Allows access to an approximate location.
android.permission.ACCESS_BACKGROUND_LOCATION	Allows a to access the location when running in the background.
android.permission.READ_PHONE_STATE	Allows read-only access to phone state, including current network information, the status of all ongoing calls, and a list of other mobile phone accounts.
android.permission.ANSWER_PHONE_CALLS	Allows to answer incoming phone calls.
android.permission.READ_CONTACTS	Allows reading the phone's contacts.
android.permission.SYSTEM_ALERT_WINDOW	Allows displaying windows on top of other apps or the system.
android.permission.CAMERA	Allows access to mobile phone camera.
android.permission.READ_CALENDAR	Allows reading the user's calendar events and details from the device's calendar.
android.permission.SEND_SMS	Allows sending Short Message Service (SMS) messages from the device.
android.permission.CALL_PHONE	Allows initiating phone calls from the device without requiring confirmation.
android.permission.READ_CALL_LOG	Allows reading the call log from the device.
android.permission.GET_ACCOUNT	Allows access to the list of accounts in, the Accounts Service, which contains the list of accounts that are associated with the user.
android.permission.AUTHENTICATE_ACCOUNTS	Allows to act as an authenticator for the accounts on the device.
android.permission.AUTHENTICATE_ACCOUNTS	Allows to perform operations such as adding and removing accounts.

Listing 4: Network configuration

```xml
<domain-config cleartextTrafficPermitted="true">
    <domain includeSubdomains="true">garmin.com
    </domain>
    <domain includeSubdomains="true">garmin.cn
    </domain>
    <domain includeSubdomains="true">garmincdn.com
    </domain>
    <domain includeSubdomains="true">strava.com
    </domain>
    <domain includeSubdomains="true">127.0.0.1
    </domain>
    <domain includeSubdomains="true">localhost
    </domain>
</domain-config>
```

CERTIFICATE ANALYSIS focuses on the application's security Certificates and signatures. Here, it found a high-severity problem. MobSF describes the finding as:

Application is signed with MD5. MD5 hash algorithm is known to have collision issues. Android requires that all applications be digitally signed with a certificate before being installed or updated. Garmin Connect uses `MD5` to sign the certificate. This can be a major or minor problem depending on whether `MD5` is the only method of hashing used (sometimes `MD5` is used as a backup in case of country restrictions with other ways). It can become dangerous if that's the case since an attacker could brute-force the signature. When the malicious user discovers it, he/she could create a customized application with malicious code, sign the certificate with the same signature, and publish the application to exploit unaware users that install it. This practice is becoming increasingly typical.

MANIFEST ANALYSIS is related to the analysis of the Android Manifest. Every Android project must have an XML file called `AndroidManifest.xml` at the root of the project source set. The manifest file describes essential information about the application to the Android build tools, the Android operating system, and Google Play.

The Android Manifest is a good entry point for dynamic analysis since it can discover permissions or other configuration flaws in a specific component (Li et al., 2016). MobSF found no issues classified as high. It only found six medium results related to components shared with other applications (Android:exported=true). However, the developers defined permissions to secure these components, so it is not an issue.

CODE ANALYSIS MobSF analyzes the application's code in search of coding flaws and relates them to Common Weakness Enumeration (CWE) and the OWASP Top 10. The scanner highlights the files related to the finding and the specific code line. MobSF identified 17 issues. However, after analyzing the code files where these issues were generated, we could narrow this down to 6 noteworthy issues to explore. The others are not considered helpful for the dynamic analysis or are related to external dependencies such as libraries.

This App uses SSL certificate pinning to detect or prevent MITM attacks in secure communication channels. The first issue found is not an issue but an application security implementation, so it was classified as **secure**. SSL pinning prevents dangerous and complex security attacks. This security measure pins the identity of trustworthy certificates on mobile apps and blocks unknown files from suspicious servers. In the dynamic analysis, we will further analyze and explore this feature.

The App logs information. Sensitive information should never be logged. Although it is not a vulnerability, the second issue can become problematic, so MobSF classified the severity as **info**. Developers usually log information to the console during development for various purposes, such as debugging. However, when the application enters the production stage, all log functions should be deactivated since attackers can exploit them to identify possible messages with private data or find ways to use features. We already saw in the file analysis that the application logs personal information related to the HTTP communication, such as the OAuth 1 and 2 data.

The App uses the encryption mode CBC with PKCS5/PKCS7 padding. This configuration is vulnerable to padding Oracle attacks. The scanner classifies the third issue as **high** severity. Padding is a common practice in cryptography, consisting of adding data to a message before it is encrypted. Oracle Padding is an attack in which a function decrypts messages and leaks the padding, allowing attackers to read sensitive data or escalate their privileges without knowing the key used in cryptographic operations. The attack is commonly associated with the Cipher-Block Chaining (CBC) encryption mode, which encrypts data only in blocks of specific sizes, so it is necessary to use padding for the correct functioning of its algorithm. The padding schemes commonly used in these operations are PKCS5 and PKCS7 (Bardou et al., 2012). Listing 5 shows a Java function from the application that is vulnerable to this type of attack.

Insecure WebView Implementation. Execution of user-controlled code in WebView is a critical Security Hole. The next issue is classified as **warning** by the scanner and

Listing 5: Java file flagged as having a vulnerability

```java
public static byte[] g(String str, String str2, byte[] Barr) {
    try {
        SecretKeySpec secretKeySpec = new SecretKeySpec(str.getBytes(),
        ↪  "DESede");
        Cipher cipher = Cipher.getInstance("DESede/CBC/PKCS5Padding");
        cipher.init(2, secretKeySpec, new IvParameterSpec(m.d(cipher, str2)));
        return cipher.doFinal(Barr);
    } catch (Exception unused) {
        return null;
    }
}
```

is related to WebView. We already identified in the file analysis that the application uses WebViews. Since they are browsers run inside the application, they can be used as a vector of attack by injecting malicious javascript code that will be run by the browser, possibly infecting the application.

App uses SQLite Database and executes raw SQL query. Untrusted user input in raw SQL queries can cause SQL Injection. Also, sensitive information should be encrypted and written to the database. The fifth issue detected by MobSF is of **medium** severity. It detected that the application executes raw SQL queries inside the code functions. One of the most common types of attacks is SQL injections, where an attacker sends in a text input raw SQL commands to obtain data or response from the database. These attacks typically bypass authentication or retrieve user information [33].

This App may have root detection capabilities. The last issue we will discuss is another security feature that the scanner detected. However, it is a false positive, or the feature is poorly programmed since we could use the application on a rooted phone. However, MobSF also identified that the application detects virtual machines and, indeed, the application does not run on virtual machines.

RECONAISCANCE .MobSF also collects hard-coded information such as Internet Protocol (IP) addresses, URLs, database addresses, secrets, and more. From the information recovered by MobSF we highlight the following: the applications **firebase** database [34]. The privacy trackers, found in the code, MobSF found trackers related to **Facebook** and **Google Analytics**. And a list of all possible secrets that are hard-coded in the application such as passwords, API keys and tokens. Table 17 highlights the secrets found by the scanner, they were all API keys related to third

33 https://developer.android.com/topic/security/risks/sql-injection
34 https://garmin-connect-mobile.firebaseio.com

Table 17: Garmin Connect Secrets

SECRET	VALUE
`baidu_map_apikey`	zHV...(a total of 24 characters)
`com.google.firebase.crashlytics.mapping_file_id`	437...(a total of 33 characters)
`google_api_key`	Alz...(a total of 39 characters)
`google_client_secret`	r_e2...(a total of 22 characters)
`mapv2_apikey`	Alz...(a total of 39 characters)

party services used by the application and contain little value. The other data from the list are useless strings or paths for the secure location of the data.

3.5 SUMMARY

We conclude the first part of this project and this analysis, focusing on the static analysis of the Garmin Connect application. This process allowed us to discover various facts about the application and things that can be used in the following steps: developing the ALEAPP plugins and the dynamic analysis. The Garmin Connect application stores data exclusively in the private directory. We also saw that the application has various extensive SQLite3 databases. Nevertheless, only a few databases had forensic value. Lastly, MobSF helped us better understand how the code works under the hood and possible flaws and security implementations that can be used in our favor. In table 18, we present a summary of the main static analysis findings.

Table 18: Static Analysis Findings

TYPE	FINDINGS
XML and Text files	User Profile Information Application Logs Facebook API Data Device Information Authentication Information API Information Firebase Information Encrypted Data
Databases	User Profile Data Health Values Activities Data Device Information Phone Notification GPS Location Daily Data Synchronization Logs
MobSF	Security Score Network Flaws Coding Flaws Trackers Hard coded Secrets Security Implementations

4 DYNAMIC ANALYSIS

In this chapter, we continue analyzing the Garmin Connect application. We use all the knowledge acquired during the background study and the post-mortem analysis to execute a dynamic application analysis. For that, we follow the steps listed in the book **Learning Android Forensics** (Skulkin et al., 2018):

1. Running the application and observing its behavior
2. Monitoring network traffic
3. Inspecting the application's user interface
4. Identifying and analyzing any potential vulnerabilities
5. Attempting to bypass any security features present in the application
6. Inspecting the application's code and resources using tools such as decompilers and debuggers
7. Identifying any sensitive data stored in the application or transmitted over the network
8. Attempting to crash the application to identify any stability issues

Some steps were already done or touched upon in the static analysis, such as observing the application behavior and vulnerabilities with the help of MobSF.

4.1 TOOLS

We needed a set of different tools for this analysis. We use ADB to interact with the phone like before. For analyzing the source code of the application, we resorted to **jadx** [1], a tool for decompiling applications and transforming `dex` code in readable `Java` code. Since Garmin Connect requires an internet connection to work, this analysis focuses on the network communication and the applications API and how it exchanges data with the server. One of the most popular tools for analyzing

1 https://github.com/skylot/jadx

Table 19: Dynamic Analysis Tool

TOOL	VERSION	USE
ADB	33.0.1	Interacting with Phone via Command Line
Frida	15.2.2	Dynamic instrumentation toolkit
jadx	1.4.5	Application decompiler
HTTP Toolkit	1.12.2	Intercepting and viewing HTTP Requests
mitmproxy	9.0	Intercepting and modifying HTTP Requests
Postman	10	Tool for testing and building API

HTTP communications is **HTTP Toolkit** [2], an open-source tool for intercepting and analyzing HTTP requests. The HTTP Toolkit application also has a feature for modifying HTTP requests and responses. Unfortunately, this is a paid feature, so we combined this tool with another tool called **mitmproxy** [3]. Mitmproxy is a command line tool for intercepting HTTP requests. The difference is that with mitmproxy, we can load Python scripts to interact with the data flow, such as dumping information or changing the requests or responses on the fly before reaching their destination. Another tool used to analyze the API was **Postman** [4]. Postman is a prevalent tool in software development to aid developers in creating and testing their API. The last tool used is called **Frida** [5]. Frida is a widely used tool in penetration testing for injecting and altering the application flow in real-time. Various Frida scripts are publicly available to test different kinds of features in an application. We will use Frida as the need arises. In the Table 19, we list all tools and their versions used during this analysis.

4.2 NETWORK ANALYSIS

Since we already had analyzed the behavior of the application, we decided to test run and complete various tasks such as:

- Authenticating in the application
- Synchronizing the application with the Vivosmart 4

2 https://httptoolkit.com/
3 https://mitmproxy.org/
4 https://www.postman.com/
5 https://frida.re/

- Uploading and updating data on the application
- Viewing different activities

We need the proxy to intercept the network traffic of an Android device. The proxy acts as man-in-the-middle between the Android device and the servers that it connects. There are several ways to accomplish network traffic interception:

- Using a proxy on a computer (mitmproxy)
- Using a fake VPN on Android to act like a proxy (HTTP Toolkit)

Using HTTP Toolkit is the more straightforward method of intercepting the network traffic. The downside is that various features are exclusive to the paid version of the application. Mitmproxy is more complex to set up but offers more packet analysis flexibility. Also, we can create `Python` scripts and add them to mitmproxy to inject or download data from specific requests and much more. The downside is that all Android traffic is routed through the proxy by default, and finding the packets related to a single Application is more challenging. We will start using HTTP Toolkit, and when the need arises, we will also use mitmproxy.

We needed to install the application on the phone and our computer to use HTTP Toolkit. HTTP Toolkit automatically adds his digital certificate to the Android phone, which means we can automatically intercept HTTPS requests of the application (except in some cases where we need an extra step that we will talk about further in this section). After starting the interception on HTTP Toolkit, we began navigating through the application to get a vast quantity of packet interception.

After using all Garmin Connect available features during the interception, we discovered that the application communicates with several hosts shown in the Table 20. Nevertheless, there could be more since we could not test all the application functionalities with Vivosmart 4.

Using the online tool **Netcraft**[6], we managed to discover the corresponding `IP` addresses. From the report generated by Netcraft, we could identify that the `URLs` are hosted on Cloudflare.

Cloudflare is a major content delivery network (CDN) service provider and is an intermediary between the website's server and the end user. When a website uses Cloudflare services, client requests are routed through Cloudflare's global network, allowing for faster content delivery and increased security. This means we cannot get the physical location from the Garmin Connect servers.

6 `https://sitereport.netcraft.com`

Table 20: Garmin Connect Hosts

HOST	USE
`api.gcs.garmin.com`	To get weather data for current location
`connect.garmin.com`	To store cache data of the application
`connectapi.garmin.com`	For the applications' API access
`diauth.garmin.com`	For the OAuth process
`omt.garmin.com`	For the synchronization process
`services.garmin`.com	To acquire the session token
`sso.garmin.com`	For user's authentication

Out of all identified hosts, the two most relevant are `sso.garmin.com` and `connectapi.garmin.com`. The first handles the authentication process for the user login. Single Sign On (SSO) refers to an authentication mechanism that enables users to access multiple applications or systems using a single set of login credentials. With SSO, users only need to authenticate themselves once and can then gain access to multiple applications without the need to provide their credentials again for each individual application.

The second host, `connectapi.garmin.com`, serves as the Garmin API host. It is responsible for retrieving all the data displayed in the application and facilitating the transmission of data to the server.

4.2.1 *Authentication*

The application login process is lengthy, so we created a diagram exemplifying the approach between the Android application and the remote server, as shown in Figure 15. The process starts when the user clicks the sign-in button, and the login screen appears.

FIRST REQUEST: After clicking the login button the request `1` will happen. Specifically, the application sends a `GET` request to the server and receives a response in HTML code to create the login dashboard. With this request, we discovered that the authentication page is not a regular activity created with XML. Instead, it is a WebView created based on the code received from the server.

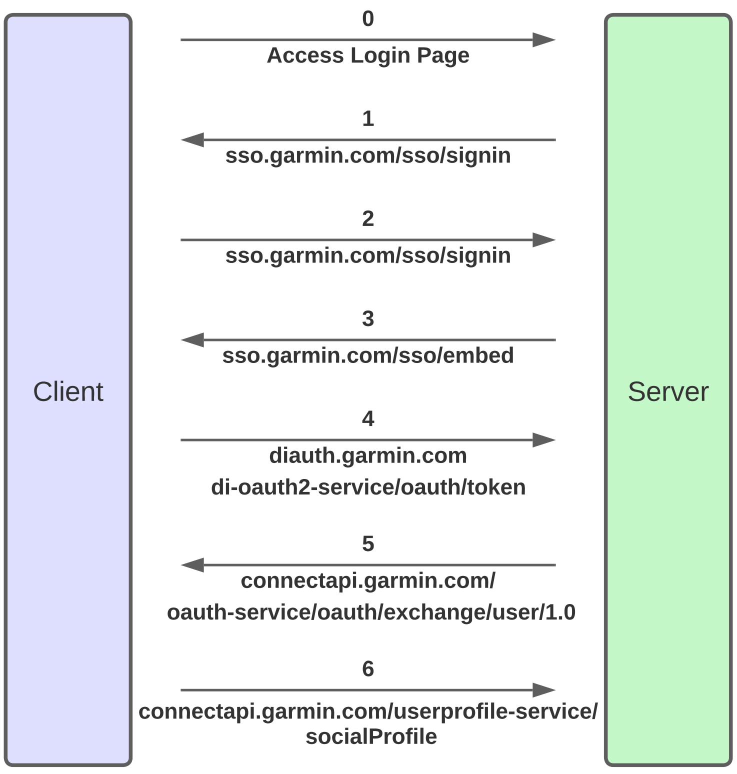

Client
Server
0
Access Login Page
1
sso.garmin.com/sso/signin
2
sso.garmin.com/sso/signin
3
sso.garmin.com/sso/embed
4
diauth.garmin.com
di-oauth2-service/oauth/token
5
connectapi.garmin.com/
oauth-service/oauth/exchange/user/1.0
6
connectapi.garmin.com/userprofile-service/
socialProfile

Figure 15: Login communication process

The presence of a WebView in the application can be considered a potential security vulnerability. It allows the execution of `JavaScript` code, which can potentially be exploited to manipulate or bypass existing security measures in the application. To assess the resilience of the WebView, we conducted several analyses utilizing tools such as `mitmproxy` and Frida. Initially, we employed Frida to enable WebView debugging, enabling us to mirror it on our computer using Google Chrome. This process is possible because Google Chrome utilizes ADB to mirror WebViews for debugging purposes. We used the Frida script **Enable WebView Debugging** [7] to enable this mode.

Frida is a dynamic code instrumentation toolkit that permits the user to inject snippets of JavaScript into native applications on Android and more. Frida hooks into the running process of the application and modifies the code on the fly without any requirement for re-launching or re-packaging.

We used Frida version 15.2.2, we also needed to install the corresponding version of Frida-tools which is 11.0.0. Frida-tools can be installed with Python `PIP` as follows:

7 https://codeshare.frida.re/@gameFace22/cordova---enable-WebView-debugging/

```
pip install frida==15.2.2
pip install Frida-tools==11.0.0
```

To run Frida, we must install it on the end device, in this case, the smartphone. For that, one needs to download the Frida Server from Frida's Github [8] and upload it to the device using ADB with the command:

```
adb push ./frida-server-15.2.2 /sdcard/Download/
```

After that, we need need to activate it as root using the following commands:

```
adb shell
su
cd /data/local/tmp
cp /sdcard/Download/frida-server-15.2.2 .
chmod 755 frida-server-15.2.2
./frida-server-15.2.2 &
```

The user needs to reactivate the Frida server if the phone reboots. Now we can use Frida to inject scripts into Android applications. Below we show how to execute the script on a computer connected by USB to the smartphone, to enable WebView debugging:

```
frida -U --no-pause --codeshare
↪  gameFace22/cordova---enable-WebView-debugging -f
↪  com.garmin.android.apps.connectmobile
```

After, we could open the WebView on our computer and use the Google Chrome developer tools to explore the HTML code. Listing 6 shows the Javascript code that is responsible for the login validation. Using that, we discovered that it uses the `JQuery` library for validating users' input.

SECOND REQUEST: After the user fills out the authentication form, the data will be sent to the server in a `POST` request. If the response is positive with a status code of 200, the application continues the process with the server. The client receives HTML code again as a response to the request, and this code presents the loading view before continuing. This code also contains a token for use in the next step. We tried modifying the `JQuery` code of the first request to disable the form validation.

8 https://github.com/frida/frida/releases

Listing 6: Partial Javascript code for the login page

```
consoleInfo("signin.html result: [" + result + "]");
consoleInfo("signin.html embedWidget: [" + embedWidget + "],
↪  createAccountConfigURL: [" + createAccountConfigURL + "], socialEnabled: ["
↪  + socialEnabled + "], gigyaSupported: [" + socialEnabled + "],
↪  socialConfigURL(): [" + socialConfigURL + "]");
    jQuery(document).ready( function() {
        jQuery("#username").rules("add",{
            required: true,
            messages: {
                required:  "Email is required."
        }}});

        jQuery("#password").rules("add", {
            required: true,
            messages: {
                required:  "Password is required."
            }
        });
        jQuery("#password").on('input', function () {
            if (jQuery("#passwordToggle").html() == "Hide") {
                jQuery(this).attr("type", "text");
                jQuery("#passwordToggle").html(jQuery("#passwordToggleHide").val
                ↪  ());
            } else if (jQuery("#passwordToggle").html() == "Show"){
                jQuery(this).attr("type", "password");
                jQuery("#passwordToggle").html(jQuery("#passwordToggleShow").val
                ↪  ());
            }
        });
    });
```

Nevertheless, it did not make a difference since the view sends a `POST` request to the API when it responds.

We decided to try other types of analyses, such as a MITM. To view and interact with the HTTP traffic, we used **mitmproxy** since we could create Python Scripts to interact with the data flow. To use mitmproxy, we needed to install it on our computer and the digital certificate on the phone. Then, the only step left is to define our computer as a proxy in the phone's network settings. Using this tool, we found out that Garmin Connect uses SSL pinning.

`SSL` pinning is a security measure where an application trusts only specific digital certificates or public keys rather than relying on trusted authorities. It helps prevent attacks involving fraudulent certificates and `MITM` (Ramírez-López et al., 2019). To bypass this feature, we used Frida with a script created by **akabe1** in Codeshare that lets users bypass multiple types of `SSL` pinning methods [9]. To execute this script, we used the command:

9 https://codeshare.frida.re/@akabe1/frida-multiple-unpinning/

```
frida -U --no-pause --codeshare akabe1/frida-multiple-unpinning -f
↪  com.garmin.android.apps.connectmobile
```

After that, we could see the traffic in mitmproxy like in HTTP Toolkit. To test against SQL Injections, we altered the password sent by the user before it hit the server using mitmproxy and the script shown listing 7:

Listing 7: Script to inject SQL

```
def request(flow: http.HTTPFlow) -> None:
    if flow.request.method == "POST":
        if "https://sso.garmin.com/sso/signin" in
        ↪  flow.request.pretty_url:
            if "password" in flow.request.text:
                flow.request.urlencoded_form["password"] = "' or '1'='1"
```

Based on the `mitmproxy` output, it was evident that the server effectively detected the invalid request and responded with an error. This outcome signifies that the API is secure and resilient against such attacks.

THIRD REQUEST: After the user logs in with the correct credentials, the application will send a `GET` request for the URL shown in step **3**, the request also contains as parameter a **Service Ticket** that the application received in the previous request. Opening the URL in the browser, we had access to the documentation for that endpoint.

Upon examination, we discovered that accessing the login form through a web browser was feasible. This led us to authenticate using the appropriate credentials, which subsequently redirected the browser to a blank page containing the **service ticket** token. This token holds significance as it will be utilized in the subsequent request illustrated in Figure 15.

FOURTH REQUEST: The application initiates another `POST` request, which is directed towards the OAuth server in order to generate the user access token for the API. Within this request, the application includes the previously received **Service Token** in the response. In return, the server provides the corresponding **Bearer Token** and **Refresh Token**, which the application can utilize for making subsequent requests. It has also been discovered that a new bearer token can be obtained by using the refresh token as a parameter instead of the service ticket. However, it is essential to note that the refresh token must still be valid for this approach to work. This process implies that only access to the Service Ticket is required to obtain the current token for the user. This particular characteristic could potentially result in a security vulnerability.

For instance, an attack scenario involves the creation of an iframe from the sign-in page, which is then added to a malicious webpage to mimic a legitimate login page. In this case, an attacker designs a fraudulent login page equipped with a keystroke logger to capture victims' credentials. Another prevalent form of attack is called **clickjacking**, wherein an attacker overlays an `HTML` form onto the iframe, making it invisible or applying the same styling as the iframe page. The user is deceived into entering their credentials into the form, unknowingly transmitting the data to a malicious server for potential storage. It is worth noting that iframe attacks have become uncommon due to the implementation of the `CORS` (Cross-Origin Resource Sharing) policy by modern browsers, which restricts JavaScript execution within elements sourced from different domains (Zhu, 2021). Nonetheless, developers can still take preventive measures to mitigate such risks, such as allowing the login page to be exclusively rendered within an iframe located on the same domain as the hosting page.

We also tested this endpoint against possible `XSS` (Cross-site scripting) by injecting in the URL Javascript. The server detected and blocked the request.

FIFTH REQUEST: Next, the application will utilize the received bearer token to initiate another `POST` request, targeting the application. This time, the request is directed towards the URL illustrated in Figure 15 at step `5`. This request aims to communicate with a different OAuth service employed by Garmin, specifically the OAuth 1 service utilized by their API and partner application. It should be noted that this service is considered legacy and possesses lower security measures in comparison to OAuth 2. Unlike OAuth 2, where the bearer token has a limited lifespan of a few hours, the credentials in OAuth 1 remain fixed. In response to the request, the server returns the user's **token** and **secret**. As the traffic is encrypted, we couldn't identify any vulnerabilities.

SIXTH REQUEST: Finally, the application issues a `GET` request to the API to retrieve the required user information for accessing the main dashboard.

Ultimately, through various analyses, we have validated the security of the authentication process. Upon successful authentication using the correct credentials and a unique token, the host `diauth.garmin.com` provides the generated OAuth token, which is used to request the session token for subsequent requests within the ongoing session. The only vulnerability identified pertains to the URL of the authentication page, as it can be rendered on any domain.

Finally, by utilizing the generated access token, we gained access to a critical artifact. Employing this token as a Bearer Token within the authorization header, alongside the OAuth 1 consumer key in the request body, we successfully executed a `POST` request to the URL `/connectapi.garmin.com/oauth-service/oauth/exchange/user/1.0`. As a result, we obtained the OAuth token and secret. It is crucial to highlight that this discovery presents a potential security risk since, with this information, we possess three of the four required parameters for requesting access on behalf of that user (consumer token, access token, and token secret). An attacker could use brute-force tactics to deduce the consumer's secret. Since OAuth 1 tokens do not expire, it would only be a matter of time until unauthorized access is obtained, enabling the attacker to extract data using the OAuth 1 credentials.

4.2.2 *Remote Extraction*

After successful authentication, we used the various resources available within the application to comprehensively understand its functionality by examining the captured requests within the HTTP Toolkit.

GET REQUESTS The first thing we noted was that application only uses the local databases for caching information. All the data shown in the different views come from `GET` requests to the API. Even when returning to other opened views, the application always refreshes the User Interface (UI) with data from the server. For example, we used `mitmproxy` to intercept a `GET` request and change the data before it reaches the application, effectively changing the data displayed to what we wanted.

During this analysis, we discovered that the access token generated after the Authentication process was sufficient to retrieve any user-specific data stored by the application. Consequently, it became apparent that having a valid user access token enabled access to all data associated with a particular user within Garmin Connect, even without user credentials or direct access to the application. This encompassed a wide range of data, including **activity data** (such as calories, time, coordinates, steps, etc.) as well as **health data** (such as heart rate, oxygen levels, stress, and sleep information). This token is valid for 24 hours.

Table 21: Data Endpoints

URL	USE
`wellness-service/wellness/dailySleepDataCharts`	Sleep Data
`mobile-gateway/snapshot/timeline/v2/forDate/`	Daily User Data
`activitylist-service/activities/search/activities`	Activities Overview Data
`activity-service/activity/`	Specific activity data
`usersummary-service/stats/stress/daily/`	Stress Data

The endpoints required for retrieving specific data may differ. However, the host `connectapi.garmin.com` remains constant throughout. Table 21 provides an overview of the endpoints identified during our analysis.

Extracting data from the API using Postman is a straightforward process. However, we aimed to develop an automated method for retrieving data from a device. This would benefit researchers and forensic practitioners seeking to obtain data from a specific phone without relying on tools like Postman or HTTP Toolkit. We created a Python script called **GCA-Extractor** (Garmin Connect API Extractor) to fulfil this objective. With this script, users can specify the desired data type and timeframe for extraction. It is important to note that users still require an access token to make API requests. Combining the findings from our post-mortem analysis with the network analysis, we devised a process that acquires a usable access token before making the request. Figure 16 illustrates an example of the script's workflow for extracting sleep data within a specified period.

First, the smartphone needs to be connected to a computer via USB. After that, it uses ADB to read the `app.log` file in Garmin's private directory, obtain the access token's last instance as shown in Listing 2, and save it in a text file. Then based on the data type chosen and timeframe, it will use Python's **http.client** to make a `GET` request to the API and obtain the data. Lastly, it will save the output in a JSON file for further analysis. The user can also add the token manually, skipping the ADB extraction process. To execute the script, the user needs to type the following:

```
python3 apiExtractor.py -a <API-key> -s <START_DATE> -e <END_DATE>
```

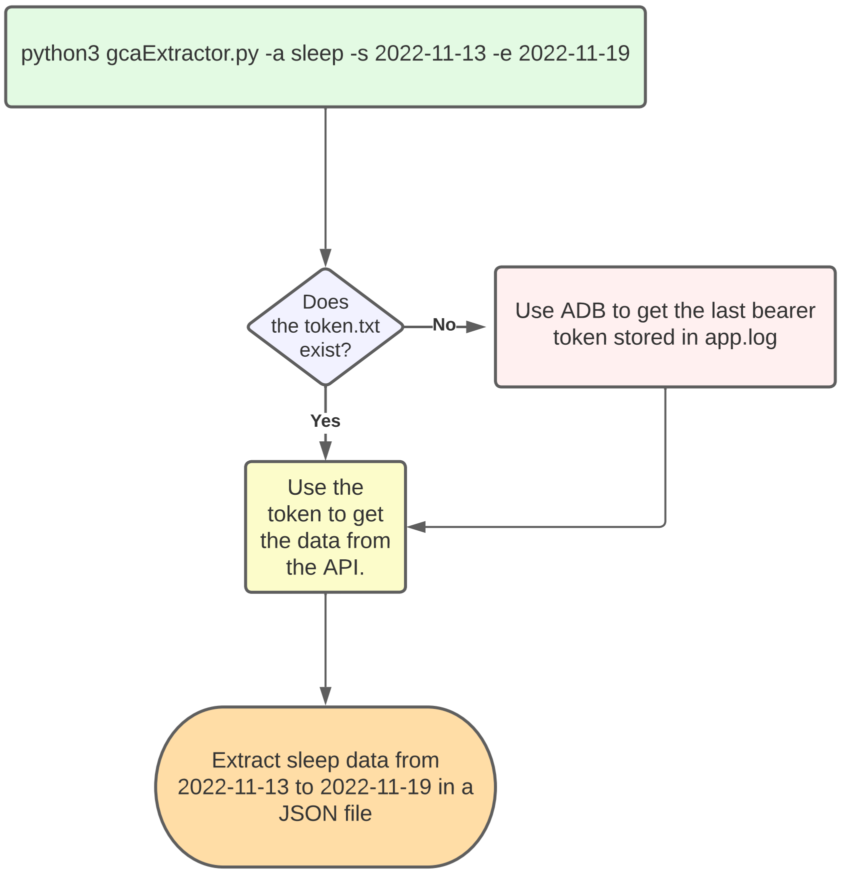

python3 gcaExtractor.py -a sleep -s 2022-11-13 -e 2022-11-19
Does the token.txt exist?
No
Use ADB to get the last bearer token stored in app.log
Yes
Use the token to get the data from the API.
Extract sleep data from 2022-11-13 to 2022-11-19 in a JSON file

Figure 16: Example of API Script Execution

Table 22: `POST` Requests

URL	USE
`activity-service/activity`	Upload Activities
`conversation-service/conversation/comment`	Write Comments
`upload-service/upload/wellness`	Upload Device Data
`usersummary-service/usersummary/hydration`	Upload Hydration Data
`weight-service/user-weight`	Upload Weight Data
`wellness-service/wellness/dailySleep`	Upload Sleep Data

As mentioned earlier, the access token operates as a JSON Web Token (JWT) with a limited validity period of 24 hours. Consequently, there is only a brief timeframe within which the access token remains valid and can be effectively utilized.

The development of the ALEAPP modules was greatly facilitated by the inclusion of this script, as it enabled the creation of modules for retrieving API data. As long as the access token remains valid, retrieving all data associated with a user's Garmin Connect account becomes possible. The script is openly available on our GitHub repository as listed in Table 31.

Currently, it supports the extraction of the following data types:

- Activities Data
- GPS Routes
- Heart Rate Data
- Sleep Data
- Stress Data
- Steps Data
- User Daily Data

POST REQUESTS Table 22 documents a collection of `POST` requests identified during our study.

Except for the `POST` request responsible for uploading device data, the majority of the `POST` requests primarily involve user-generated content and thus possess limited forensic significance. Using HTTP Toolkit, we studied the possibility of modifying requests to send forged data to the server. The application sends data to the server in the binary protocol `FIT` seen in subsection 3.3.2. We intercepted data with `mitmproxy` and tried to decode it with the script **fitdecode** to modify it before sending it to the server. However, this request's body differs from a standard user-generated FIT file so we couldn't decode this data to continue this test.

4.3 CODE ANALYSIS

A vital step during a dynamic study is to analyze the application source code to find potential vulnerabilities and flaws that attackers may exploit to bypass security features.

Android applications are often written in Java and compiled into Dalvik bytecode. Dalvik bytecode is created by first compiling the Java code to `.class` files, then converting the Java Virtual Machine (JVM) bytecode to the Dalvik `.dex` format as shown in Figure 17. The Java bytecode is compiled into Dalvik bytecode that is executed by the Android environment, which in modern devices is called Android Runtime (ART) (Mueller et al., 2022).

The `.dex` format is very memory efficient. However, it is almost impossible to reverse engineer and find flaws in binary code, so various tools such as **JADX** convert DEX code back to Java source code.

To install jadx, we need to download the binaries from GitHub and run them in the command line with the following code:

```
    jadx -gui
```

After decompiling the application, we observed that it uses obsfucation to harden its analysis. Code obfuscation is producing an executable that has its identifiers replaced by random names, severely hardening the task of understanding the code. While the process may modify actual method instructions or metadata, it does not alter the program's output (Xu et al., 2020). By making an application much more difficult to reverse-engineer, developers can protect the application against unauthorized access, bypassing licensing or other controls and vulnerability discovery.

Garmin Connect implements two obfuscation techniques: Rename Obfuscation which is renaming the packages and classes to random letters as shown in Figure 18,

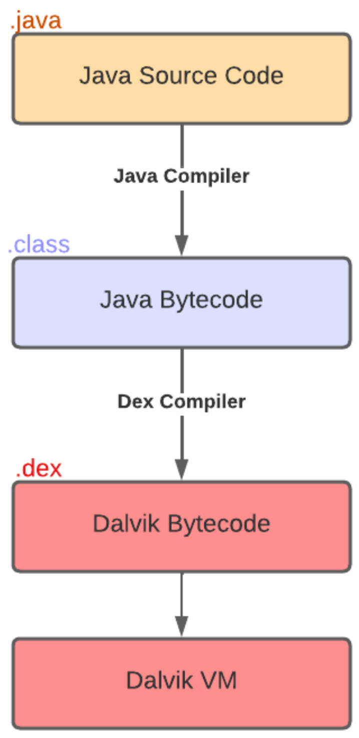


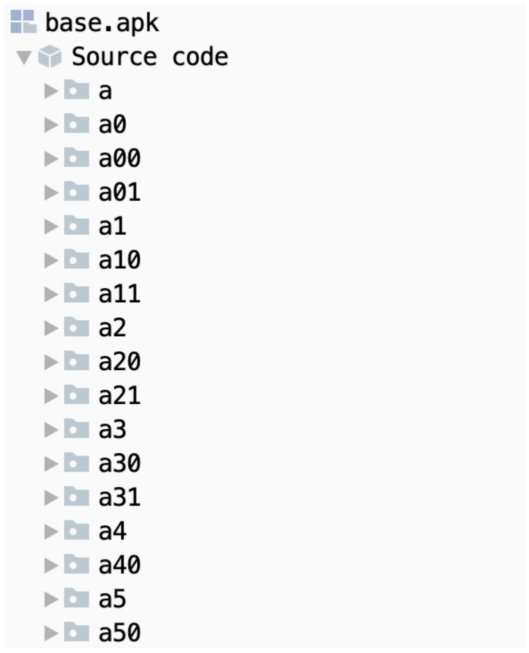
Figure 17: Android compilation process

and String codification which makes the code harder to decipher by anonymizing the name of the functions and variables as shown in Figure 19.

JADX manages to deobfuscate some code making it more readable. Nonetheless, the analysis can still be challenging, especially when the application splits its code across multiple different classes, making interactions with Frida even more difficult. Since the application performs very few functionalities locally, there is little to analyze here. We focused on searching for potential hardcoded secrets the developers may have left, but we did not find anything.

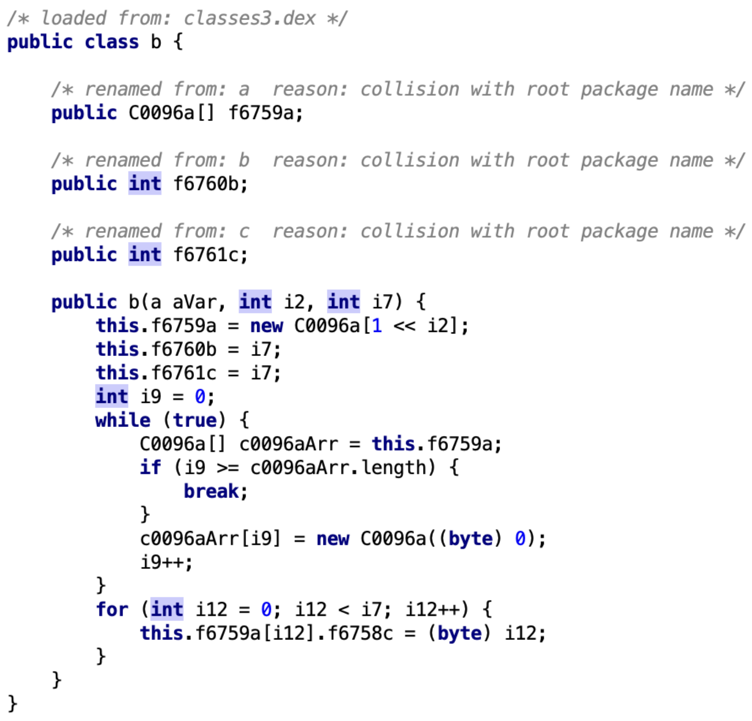
4.4 SUMMARY

We conclude this dynamic analysis without having discovered meaningful findings. This can be seen as a positive outcome, as it proves that the Garmin Connect application is secure against different kinds of attacks, as shown in the network and code analysis. We also managed to understand how the application authentication process works and how it internally fetches data. Analyzing the API helped us



```
base.apk
▼ Source code
    ▶ a
    ▶ a0
    ▶ a00
    ▶ a01
    ▶ a1
    ▶ a10
    ▶ a11
    ▶ a2
    ▶ a20
    ▶ a21
    ▶ a3
    ▶ a30
    ▶ a31
    ▶ a4
    ▶ a40
    ▶ a5
    ▶ a50
```

Figure 18: Obfuscated Directories



```java
/* loaded from: classes3.dex */
public class b {

    /* renamed from: a  reason: collision with root package name */
    public C0096a[] f6759a;

    /* renamed from: b  reason: collision with root package name */
    public int f6760b;

    /* renamed from: c  reason: collision with root package name */
    public int f6761c;

    public b(a aVar, int i2, int i7) {
        this.f6759a = new C0096a[1 << i2];
        this.f6760b = i7;
        this.f6761c = i7;
        int i9 = 0;
        while (true) {
            C0096a[] c0096aArr = this.f6759a;
            if (i9 >= c0096aArr.length) {
                break;
            }
            c0096aArr[i9] = new C0096a((byte) 0);
            i9++;
        }
        for (int i12 = 0; i12 < i7; i12++) {
            this.f6759a[i12].f6758c = (byte) i12;
        }
    }
}
```

Figure 19: Obfuscated code

Table 23: Dynamic analysis findings

BRAND	MAIN FINDINGS
Network Analysis	Authentication is done in a WebView. The application stores almost no Data locally and uses the API to get data application uses OAuth 1 and 2. With the Bearer Token, wone can get all the user's from the API. It is possible to load the login page in an Iframe. The application uses HTTPS and SSL certificate pinning. The application is secure against SQL and Javascript Injections.
Code Analysis	The code is obfuscated. There are no sensitive secrets hard-coded. The application is protected against reverse engineering.

to create a script to automate extracting data from a user using only ADB. This script is crucial for the next chapter, where we will discuss creating the ALEAPP modules for Garmin Connect. Table 23 shows what we managed to discover with this analysis.

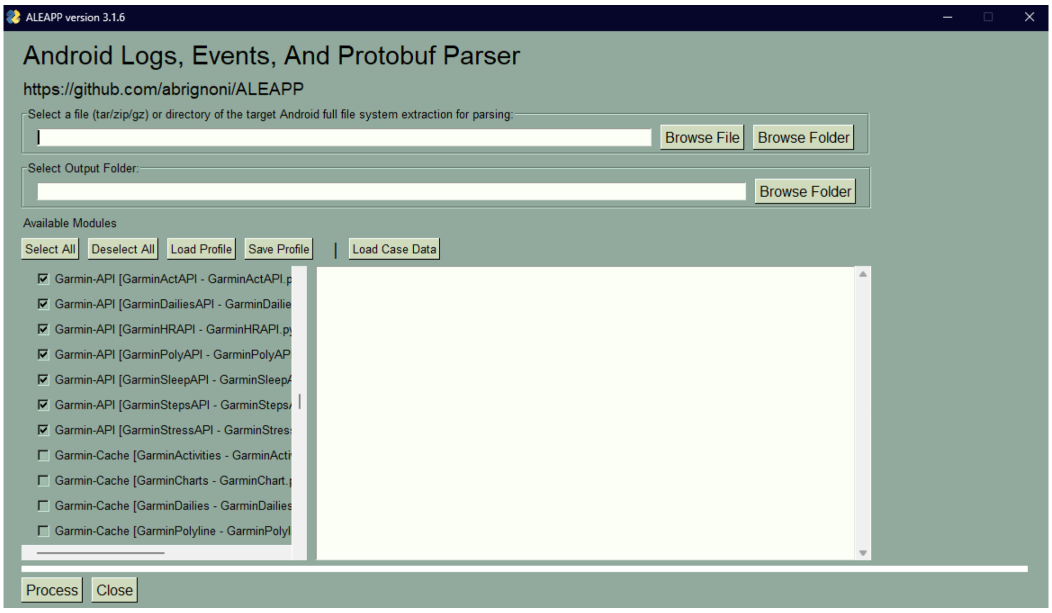
5 GARMIN CONNECT FOR ANDROID ANALYZER

We developed modules for the forensic framework ALEAPP to automate the Android Garmin Connect data analysis. We called this set of modules *Garmin Connect for Android Analyzer* or GC4AA. We also developed various new functionalities for the ALEAPP framework to augment the reports generated, present the data more interactively, or ease the report's analysis by adding improvements to existing features. In subsection 5.5, we will detail all the new features added to ALEAPP.

5.1 ALEAPP

ALEAPP or Android Logs Events And Protobuf Parser is a `Python` framework created by Alexis Brignoni Brignoni (2023) to generate a forensic report based on artifacts found in the directories extracted of Android applications. The artifacts depend on the modules selected, as shown in Figure 20. ALEAPP is a valuable triage resource within forensic laboratories, offering a crucial auxiliary solution for forensic investigators. Moreover, ALEAPP is a versatile framework enabling independent testing and evaluation of various forensic tools, further enhancing their utility and value. It is becoming a popular tool among open-source practitioners for analyzing Android applications.

Since ALEAPP is an open-source project, the creator incentivizes contributions such as creating modules for applications that are yet not supported or new features for existing ones. ALEAPP is modular and is organized so that new developers can create modules or new features for ALEAPP and seemingly integrate them. When we started developing our modules for Garmin Connect, there were no modules for Garmin Connect. We are motivated to develop these modules for ALEAPP and not create a standalone tool such as was done by Domingues et al. (2023) due to the current popularity and growth in ALEAPP from users and developers creating modules for it. Improving ALEAPP by adding a popular application such as Garmin Connect and making it available to more analysts instead of publishing a standalone tool makes sense.


ALEAPP version 3.1.6
Android Logs, Events, And Protobuf Parser
https://github.com/abrignoni/ALEAPP
Select a file (tar/zip/gz) or directory of the target Android full file system extraction for parsing:
Browse File
Browse Folder
Select Output Folder:
Browse Folder
Available Modules
Select All
Deselect All
Load Profile
Save Profile
Load Case Data
Process
Close

[✓] Garmin-API [GarminActAPI - GarminActAPI.p
[✓] Garmin-API [GarminDailiesAPI - GarminDailie
[✓] Garmin-API [GarminHRAPI - GarminHRAPI.py
[✓] Garmin-API [GarminPolyAPI - GarminPolyAP
[✓] Garmin-API [GarminSleepAPI - GarminSleepA
[✓] Garmin-API [GarminStepsAPI - GarminSteps/
[✓] Garmin-API [GarminStressAPI - GarminStres
[ ] Garmin-Cache [GarminActivities - GarminActi
[ ] Garmin-Cache [GarminCharts - GarminChart.
[ ] Garmin-Cache [GarminDailies - GarminDailies
[ ] Garmin-Cache [GarminPolyline - GarminPolyl

Figure 20: ALEAPP GUI

We created 23 different modules to extract the information from the data source. From the 23 modules, 16 parse and extract the information stored in the private directory. We called these modules *post-mortem*, Table 24 presents a brief description of each module created.

The remaining seven modules parse and extract data recovered from the API through our GCA-Extractor script. We called them API modules and presented a brief explanation of each in Table 25. Furthermore, we also developed five new features for the reports generated by ALEAPP. Our contributions were added officially in version 3.1.8 of ALEAPP [1].

5.1.1 *Installation*

To install ALEAPP one clones the repository with the following command:

```
   git clone https://github.com/abrignoni/aleapp.git
```

Since we wanted to develop code for this repository, we forked it so that we could work in our repository with source control. On the GitHub page, the developer advises using `Python 3.9` or above to work with ALEAPP.

1 https://github.com/abrignoni/ALEAPP/tree/main/scripts

Table 24: *Post-mortem* Modules

MODULE	DESCRIPTION
`GarminActivities.py`	Parses activities stored in the `cache-database`
`GarminChart.py`	Parse and display Heart Rate data from the `cache-database`
`GarminDailies.py`	Parse user daily summary from the `cache-database`
`GarminFacebook.py`	Get Information from the Facebook API
`GarminGcmJsonActivities.py`	Parse JSON activities stored in `gcm_cache.db`
`GarminJson.py`	Parse JSON data stored in `gcm_cache.db`
`GarminLog.py`	Parses data from the `app.log` file
`GarminNotification.py`	Parses data from the `notification-database`
`GarminPersistent.py`	Parse the content of the `PersistedInstallation.xml` file
`GarminPolyline.py`	Parses and converts the polyline records from the `cache-database`
`GarminResponse.py`	Parse the response cache data from the `cache-database`
`GarminSPo2.py`	Parse the SpO_2 data recorded in the `cache-database`
`GarminSleep.py`	Parses sleep data recorded in the `cache-database`
`GarminSync.py`	Parse the synchronization records stored in the in the `sync_cache`
`GarminUser.py`	Parses data from the `gcm_user_preferences.xml` file
`GarminWeight.py`	Parses weight data from the `cache-database`

Table 25: API Modules

MODULE	DESCRIPTION
`GarminActAPI.py`	Parses activity data obtained from the API
`GarminDailiesAPI.py`	Parses user daily summary data obtained from the API
`GarminHRAPI.py`	Parses heart rate data obtained from the API
`GarminPolyAPI.py`	Parses polyline data obtained from the API
`GarminSleepAPI.py`	Parses sleep data obtained from the API
`GarminStepsAPI.py`	Parses step data obtained from the API
`GarminStressAPI.py`	Parses stress data obtained from the API

To run ALEAPP, one must install the required dependencies listed in the file `requirments.txt`. This can be done using the command:

```
pip3 install -r requirements.txt
```

It is recommended to create a virtual environment in case the user has conflicting global packages installed. In Linux, one will also needs to install `tkinter` (python Graphical User Interface (GUI) package) with the command:

```
sudo apt-get install python3-tk
```

5.1.2 *Plugin Creation*

ALEAPP was built around the idea of being scalable and modular, developers only need to produce a single Python file to add it to the framework. Inside this file, we will parse the data as he sees fit and output the data in the Hypertext Markup Language (HTML) report generated by ALEAPP using the functions already developed. The plugins are stored in the folder **artifacts** inside of the **scripts** folder, Listing 8 shows the basic structure for an ALEAPP module. Simply put, an ALEAPP module does is to read the target file, extract the data and present it to the user. Since this is very basic and we added more features to ALEAPP. For that, we needed to add code to the base Python files that contain the code to generate HTML reports.

5.2 TOOLS

For this part of the project, we used the standard tools needed for software development. The modules were developed in Python 3.10, resorting to the Integrated Development Enviorment (IDE) PyCharm.

We had to implement new features in ALEAPP to create our modules. For that, he had to add various Python and Javascript libraries. Table 26 shows all the dependencies used, the version used, and the language associated with it.

Listing 8: Template Structure of ALEAPP modules

```python
import datetime
from scripts.artifact_report import ArtifactHtmlReport
import scripts.ilapfuncs

def get_cool_data1(files_found, report_folder, seeker, wrap_text):
    # let's pretend we actually got this data from somewhere:
    rows = [
     (datetime.datetime.now(), "Cool data col 1, value 1", "Cool data
     ↪  col 1, value 2", "Cool data col 1, value 3"),
     (datetime.datetime.now(), "Cool data col 2, value 1", "Cool data
     ↪  col 2, value 2", "Cool data col 2, value 3"),
    ]

    headers = ["Timestamp", "Data 1", "Data 2", "Data 3"]

    # HTML output:
    report = ArtifactHtmlReport("Cool stuff")
    report_name = "Cool DFIR Data"
    report.start_artifact_report(report_folder, report_name)
    report.add_script()
    report.write_artifact_data_table(headers, rows, files_found[0])  #
    ↪  assuming only the first file was processed
    report.end_artifact_report()

    # TSV output:
    scripts.ilapfuncs.tsv(report_folder, headers, rows, report_name,
    ↪  files_found[0])  # assuming first file only

    # Timeline:
    scripts.ilapfuncs.timeline(report_folder, report_name, rows, headers)


__artifacts__ = {
    "cool_artifact_1": (
        "Really cool artifacts",
        ('*/com.android.cooldata/databases/database*.db'),
        get_cool_data1)
}
```

5.3 *POST-MORTEM* MODULES

The *post-mortem* modules were developed by leveraging the inherent functionality of ALEAPP, which involves parsing data within Android application directories to locate artifacts and generate comprehensive reports. Using the data collected from the *post-mortem* analysis, our modules will parse the files listed in Table 27 to retrieve the corresponding artifacts.

5.3.1 *Activities modules*

There are three modules related to the activities stored in the `cache-database`: `GarminActivities.py`, `GarminChart.py`, and `GarminPolyline.py`. Next, we describe each of them.

Table 26: Libraries Used

LIBRARY	VERSION	LANGUAGE
Cal Heatmap	4.0	Javascript
Chart JS	4.2	Javascript
D3	7	Javascript
Folium	0.14.0	Python
Geopy	2.3.0.	Python
Highlights JS	11.7.0	Javascript
Moment JS	2.29.4	Javascript
Polyline	2.0	Python
Popper	2.11.6	Javascript

Table 27: Parsed artifacts

ARTIFACT	FILE
`Activities`	`cache-database` `gcm_cache.db`
Activity Charts	`cache-database`
Daily Summaries	`cache-database`
GPS Data	`cache-database`
Response Cache	`cache-database`
Sleep Data	`cache-database`
Weight Data	`cache-database`
`UserData`	`gcm_user_preferences.xml` `gcm_cache.db`
Log Data	`app.log`
Facebook	`com.Facebook.internal.preferences`
Firebase	`PersistedIntallation.xml`
Notification	`notification-database`
Synchronization Cache	`sync_cache.db`

GARMINACTIVITIES.PY This module extracts all data related to a workout activity performed by the user and uploaded to the application. The data is then stored in the tables `activity_details` and `activity_summaries`. The module presents all activities recorded and the following data related to each activity:

- ID
- Start and End time
- Name and type
- Distance, Duration, and time moving
- Elevation
- Speed
- Start latitude and longitude
- Profile Picture
- Calories Burned
- Heart Rate
- Running Cadence
- Steps
- Maximal Oxygen Consumption (vO_2)

GARMINCHART.PY This module extracts data stored in the `activity_chart_data` table. As explained earlier, in the database analysis, this table stores the coordinates to draw specific data charts in the application, such as the Heart Rate measurement collected during a workout. The goal of this module is to present the activities and associated records in the `activity_chart_data` and list them. Since the application uses these values to plot charts, we did the same in our report. We developed this feature since ALEAPP by default did not have a feature to create and present data charts. Every activity record has a button associated with it, so when clicked shows the respective chart based on the data, as shown in Figure 21.

GARMINPOLYLINE.PY This module shows the activities which are recorded in the `activity_polyline` table. The goal is to show the workout routes taken by the user. As the application only stores a polyline string and not the coordinates, we created a method for converting polylines to GPS coordinates. The process has already been explained during the database analysis (see subsection 3.3.3). Data is displayed in a table, as shown in Figure 22. Here, the user changes the route shown in the map by clicking the button **Show Map** on the desired record. The module also allows to download a `KML` file with coordinates to open the data on Google Earth. And just as in ADB Extractor, the user can also download the `XLSX` file with

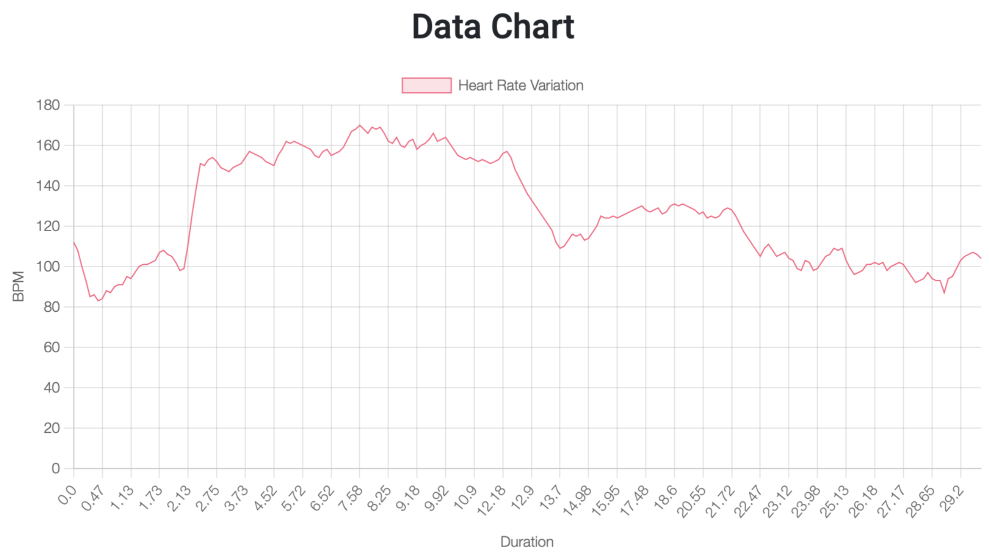


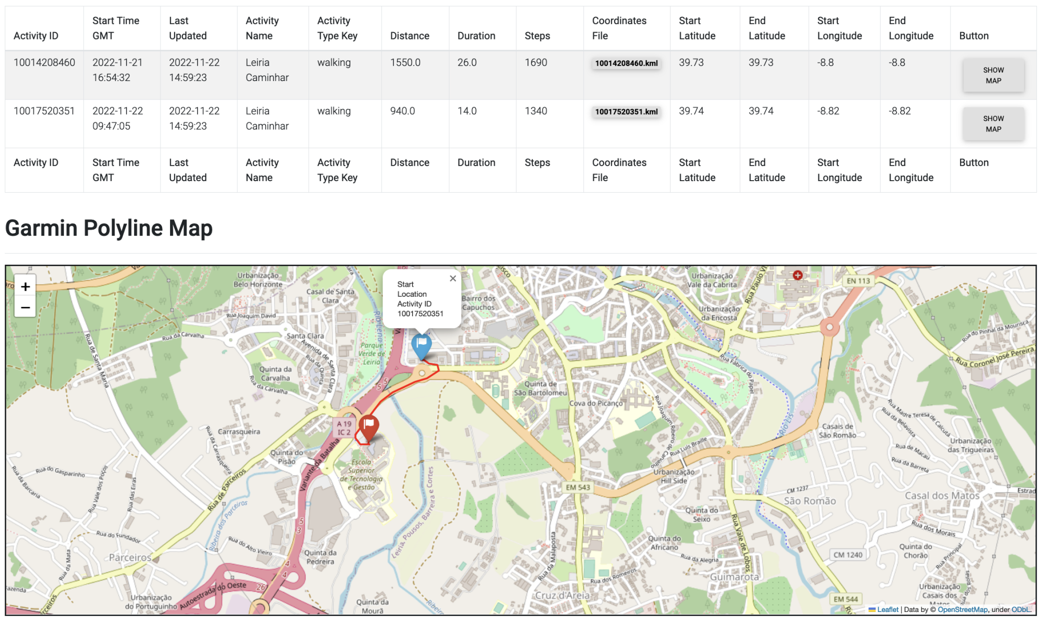
Figure 21: ALEAPP Heart Rate Chart

the coordinates and additional information obtained from `geopy`. The module can also be executed without internet connectivity, losing the ability to create the `XLSX` file, Figure 11 shows an excerpt of the Excel file generated.

5.3.2 *User Daily Module*

The module `GarminDailies.py` extracts and presents the daily summary data stored by the application in the `user_daily_summary` table. Each record represents a different day. The module presents the following data:

- Calorie Data
- Steps
- Distance
- Floors
- Heart Rate
- Stress
- Body energy level (also called body battery)
- Hydration
- SpO_2



Activity ID	Start Time GMT	Last Updated	Activity Name	Activity Type Key	Distance	Duration	Steps	Coordinates File	Start Latitude	End Latitude	Start Longitude	End Longitude	Button
10014208460	2022-11-21 16:54:32	2022-11-22 14:59:23	Leiria Caminhar	walking	1550.0	26.0	1690	10014208460.kml	39.73	39.73	-8.8	-8.8	SHOW MAP
10017520351	2022-11-22 09:47:05	2022-11-22 14:59:23	Leiria Caminhar	walking	940.0	14.0	1340	10017520351.kml	39.74	39.74	-8.82	-8.82	SHOW MAP
Activity ID	Start Time GMT	Last Updated	Activity Name	Activity Type Key	Distance	Duration	Steps	Coordinates File	Start Latitude	End Latitude	Start Longitude	End Longitude	Button

Garmin Polyline Map

Start Location Activity ID 10017520351

Figure 22: Extract of GPS Report Module

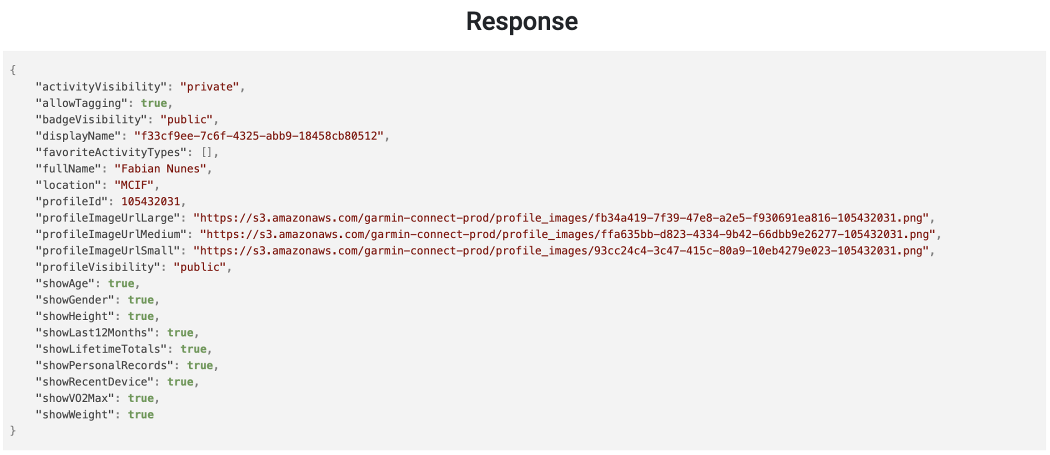
5.3.3 *Health Data Modules*

Related to the health data stored in the `cache-database`, we could extract the data related to SpO_2 readings using the module `GarminSPo2.py`, the sleep data using `GarminSleep.py` and the weight recorded in the application using the module `GarminWeight.py`.

GARMINSPO2.PY parses the information of the table `acclimation_pulse_ox_details`, where the application stores recent SpO_2 readings. The module extracts and presents the readings date/time, the average SpO_2 value, and a data chart with the SpO_2 variation based on the values stored in the table.

GARMINSLEEP.PY extracts the data from the `sleep_detail` table. This table stores the sleep records. The module extracts and displays the following information:

- Time Slept (Begin, End, and total)
- Seconds in Deep, Light, and Rem Sleep
- Awake Seconds
- Average SpO_2
- Average Heart Rate


Response

```
{
    "activityVisibility": "private",
    "allowTagging": true,
    "badgeVisibility": "public",
    "displayName": "f33cf9ee-7c6f-4325-abb9-18458cb80512",
    "favoriteActivityTypes": [],
    "fullName": "Fabian Nunes",
    "location": "MCIF",
    "profileId": 105432031,
    "profileImageUrlLarge": "https://s3.amazonaws.com/garmin-connect-prod/profile_images/fb34a419-7f39-47e8-a2e5-f930691ea816-105432031.png",
    "profileImageUrlMedium": "https://s3.amazonaws.com/garmin-connect-prod/profile_images/ffa635bb-d823-4334-9b42-66dbb9e26277-105432031.png",
    "profileImageUrlSmall": "https://s3.amazonaws.com/garmin-connect-prod/profile_images/93cc24c4-3c47-415c-80a9-10eb4279e023-105432031.png",
    "profileVisibility": "public",
    "showAge": true,
    "showGender": true,
    "showHeight": true,
    "showLast12Months": true,
    "showLifetimeTotals": true,
    "showPersonalRecords": true,
    "showRecentDevice": true,
    "showVO2Max": true,
    "showWeight": true
}
```

Figure 23: ALEAPP JSON Block

GARMINWEIGHT.PY presents all records related to the user weight that is stored in the table `weight`. It also presents a chart with the fluctuation of weight during the records.

5.3.4 *Response Module*

As analyzed before, the table `response_cache` stores a cache of request data to the API, saving both the request and its corresponding JSON response. The module `GarminResponse.py` presents this data. Since the table stores different JSON requests and responses, we decided to present the complete response in the report. To ease the understanding of the data, we created a functionality to display the JSON code with indentation and different colors, as shown in Figure 23. Each record represents a different response, and each has a button to show the respective JSON response.

5.3.5 *GCM Modules*

As studied earlier (see subsection 3.3.3), the `gcm_cache.db` stores, in JSON format, a cache of user data and activity-related data similar to the `cache-databas e`. There still can be data that is worth analyzing, so we created two modules: `GarminGcmJsonActivities.py` that will extract the JSON blocks from the table `json_activities`, and the module `GarminJson.py` and `GarminResponse.py`. The former extracts JSON blocks from the `json_activities` table, while the latter parses and extracts data from the `json` table. Both modules present the data in a similar way as the `GarminResponse.py` module.

5.3.6 *Notifications and Synchronization Modules*

Both the database `nofications-database` and `sync_cache.db` store interesting data. The first stores the phone notifications sent to the smartband, while the second holds the synchronization process of the smartband with the application. To exploit these databases, we created the modules `GarminSynch.py` and `GarminNotifications.py`.

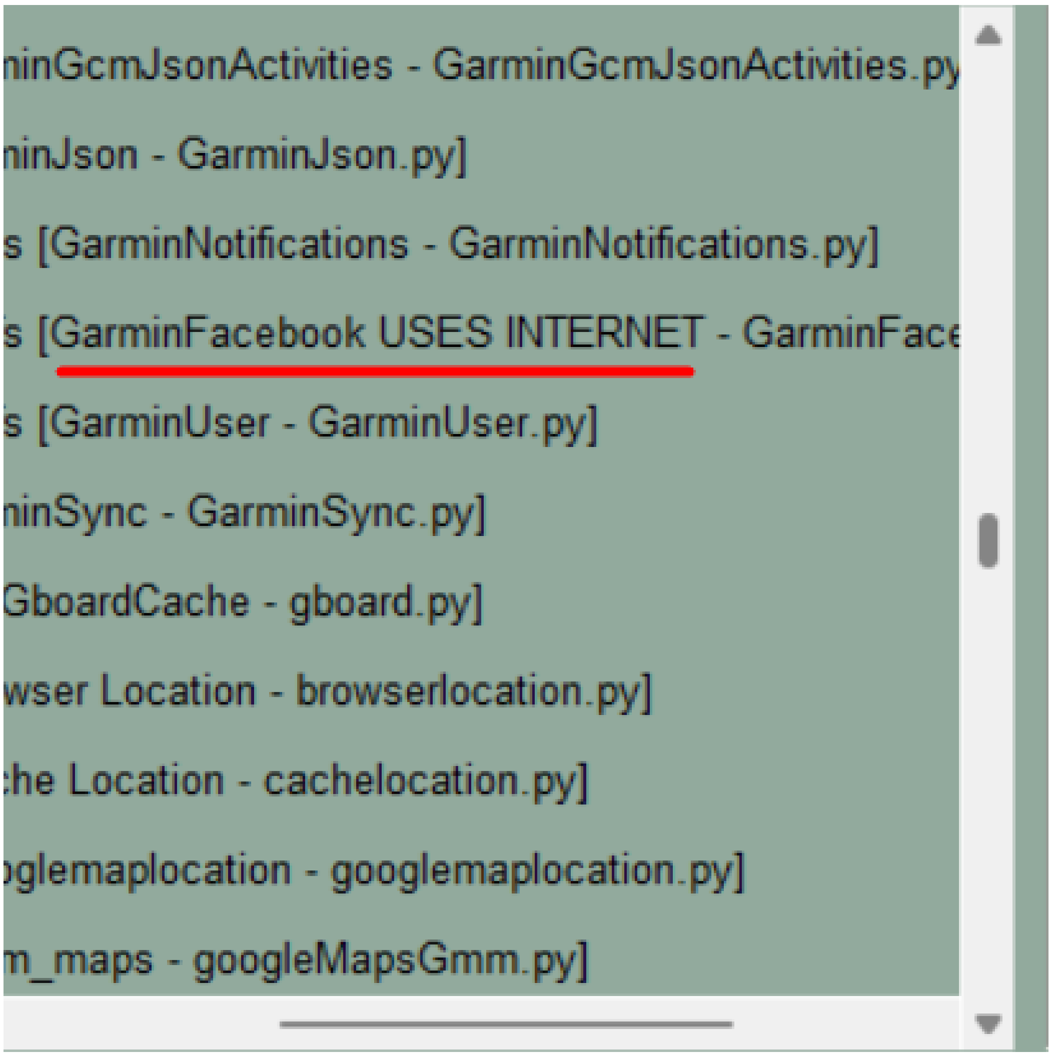
5.3.7 *Files Modules*

We also detected meaningful data stored in the `files` folder of the private directory, namely the file `app.log`, which stores a cache of HTTP requests and `PersistedInstalation.xml` that contained the access token of the Firebase database. The modules are, respectively, `GarminLogs.py` and `GarminPersisted.py`. They can extract the tokens from both files.

5.3.8 *SharedPreferences Modules*

Lastly, through the *post mortem analysis*, we found some relevant data related to the user in the `SharedPreferences` folder. Of all the files inside this directory, the most important ones are `gcm_user_preferences.xml` which has data related to the user and `com,facebook.internal.preferences.APP_SETTINGS.xml` which holds data related to the user's Facebook account in case the user connected their Garmin account to a Facebook account.

For `gcm_user_preferences.xml`, we created the module `GarminUser.py`, and for the second, the module `GarminFacebook.py`. The first is a standard module that extracts the information from the XML file and presents it in the report. The Facebook module can retrieve information from Facebook's Graph API using Python's **http.client** and the token stored in `com.facebook.sdk.USER_SETTINGS .xml`. Since the ALEAPP user may not want to connect to the internet, we added a warning in front of the module in the ALEAPP GUI to alert the user that the modules attempt to connect to the internet, Figure 24 shows the warning persented.



```
ninGcmJsonActivities - GarminGcmJsonActivities.py
ninJson - GarminJson.py]
s [GarminNotifications - GarminNotifications.py]
s [GarminFacebook USES INTERNET - GarminFace
s [GarminUser - GarminUser.py]
ninSync - GarminSync.py]
GboardCache - gboard.py]
wser Location - browserlocation.py]
che Location - cachelocation.py]
oglemaplocation - googlemaplocation.py]
n_maps - googleMapsGmm.py]
```

Figure 24: ALEAPP JSON Block

5.4 API MODULES

The typical *post-mortem* analysis performed by ALEAPP focuses on analyzing data directories extracted from the device. However, after uncovering significant insights during dynamic analysis and utilizing the `GCA-Extractor` script, we created an additional set of modules specifically designed to parse data obtained through this script. These modules are responsible for parsing the JSON files generated by the script and generating reports containing the discovered artifacts, mirroring the functionality of the existing modules. These modules cover the following data categories:

- Activities
- GPS Data
- Heart Rate Data
- Sleep Data
- Step Data
- Stress Data
- Daily Summary Data

Although these modules share similarities with the ones developed for the *post-mortem* analyses, they offer enhanced capabilities due to the extraction of data through an API. This enables us to present more detailed information, including using data charts as exemplified in Figure 21. We also extracted data not present in the *post-mortem* analysis shown in the subsections below.

5.4.1 *Heart Rate Module*

`GarminHRAPI.py`, parses the data extracted from the API related to the user's heart rate. This module shows a record for each day with the lowest, highest, and average heart rate values and a data chart with heart rate variation registered during the day.

5.4.2 *Sleep Module*

`GarminHRAPI.py` may look similar to its counterpart in the *post-mortem* modules. However, it is possible to obtain more data related to each sleeping session from the API compared to the database. It was possible to show a chart with the sleep

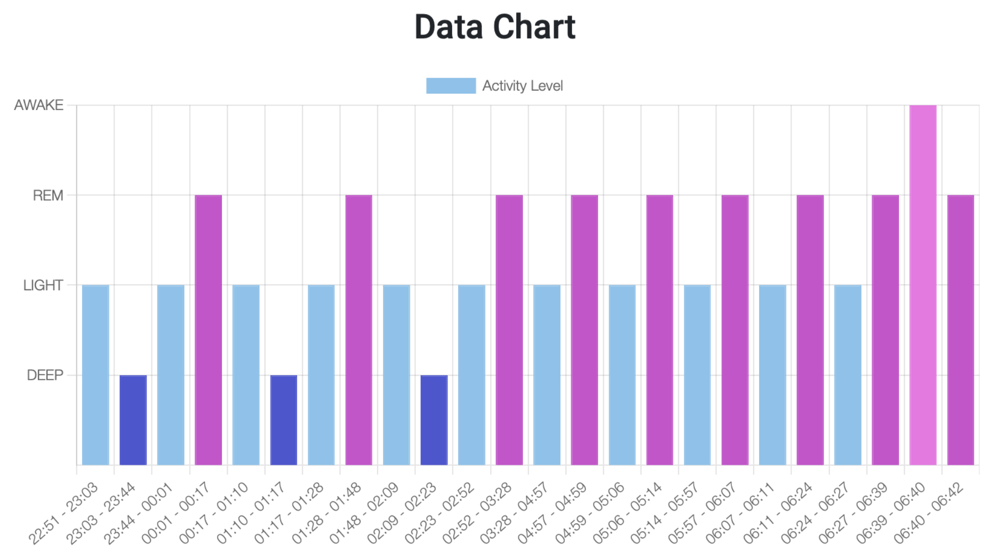


Figure 25: ALEAPP Sleep Phase Chart

phase during the different hours can be seen in Figure 25. Also, it was possible to extract the SpO_2 records during the acquisition and show it in a chart.

5.4.3 *Steps Module*

`GarminSteps.py` parses the step data from the API, where each record represents a day. It shows the number of steps taken, calories burned, and the floors ascended and descended, and the user can open a data chart showing when the users were more active during the day.

5.4.4 *Stress Module*

`GarminStress.py`, parses the stress information extracted from the API. The module reports, for each day, the stress level (from 0 to 100), and duration. The module plots a chart with the stress levels from the different days to highlight the changes.

5.5 FEATURES ADDED

ALEAPP, a powerful forensic tool, has been further enhanced during our development process to incorporate specific ideas for report presentation. We strongly

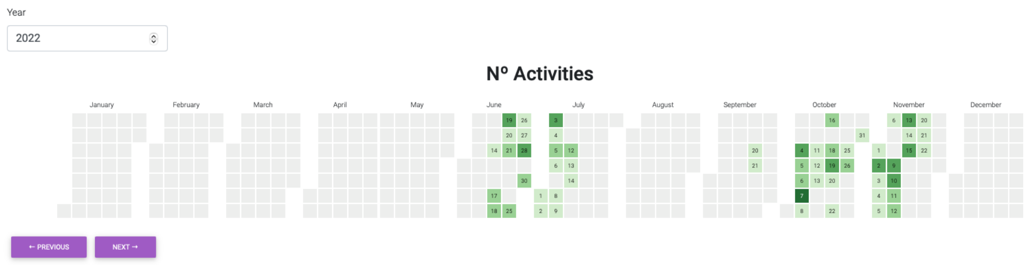


Figure 26: ALEAPP Heatmap

believe that augmenting the tool with various visual functionalities would significantly improve its overall capabilities. Leveraging the modular nature of ALEAPP facilitated the rapid implementation of these features. The following functionalities have been integrated into ALEAPP:

- Heatmap Visualization
- Date Filtering
- GPS Maps
- Data Charts
- Code Syntax Highlights

Next, we describe each of them.

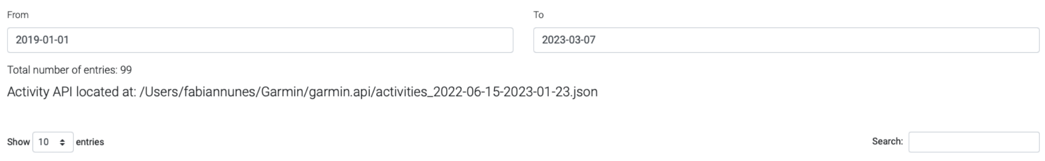
5.5.1 *HeatMap Visualization*

Heatmaps provide a valuable visual representation of data over an extended period. To enhance this capability, we developed a new feature that presents a calendar-style heatmap, enabling users to observe the frequency of records on specific days. In particular, we implemented this feature within the activities module, generating a heatmap displaying the number of activities conducted each day. Please refer to Figure 26 for an illustration of this heatmap. We developed this feature using the Javascript library Cal-Heatmap [2].

5.5.2 *Date Filtering*

The inclusion of a date filtering feature proves to be uncomplicated yet highly advantageous. Given the potential magnitude of elements within tables and the objectives of our modules, analysts can effectively filter results by specifying a date

2 https://cal-heatmap.com



From

2019-01-01

To

2023-03-07

Total number of entries: 99

Activity API located at: /Users/fabiannunes/Garmin/garmin.api/activities_2022-06-15-2023-01-23.json

Show 10 entries

Search:

Figure 27: ALEAPP Date Filter

range. This functionality, depicted in Figure 27, extends to date fields in tables, providing the user with the ability to apply filters. It is worth noting that this feature can be implemented in any module by simply specifying the desired date field of the table.

5.5.3 *GPS Maps*

One of our goals was to display GPS routes instead of just showing the coordinates, as maps proved a much better situational awareness. For this purpose, we resorted to Python `folium` library [3]. This library allows to process and visualize data within a leaflet map. This is done as follows:

1. The polyline is decoded to coordinates through the polyline library;
2. Next, through folium, an HTML map is rendered.

The ALEAPP user can also access each Polyline map in the report folder, where it can be opened and analyzed. Figure 22 shows the map generated by ALEAPP.

5.5.4 *Data Charts*

We have already shown various figures from the charts created for our modules (see Figure 21). We created these charts using the `Charts.js` Javascript library [4]. Charts ease the data analysis of specific data, such as arrays of values of a specific timespan.

3 https://python-visualization.github.io/folium/
4 https://www.chartjs.org

5.5.5 *Code Syntax Highlights*

The last feature we added to ALEAPP was custom code blocks to render the information extracted from databases. By default, HTML already has a tag called `code` for computer code. However, this tag only formats the font family of the text. We aimed to add custom blocks where the code is rendered with the correct spacing and color highlights, just like code editors. This was archived through the `highlights.js` library [5], as it color renders JSON data, the result can be seen in Figure 23.

5.6 AUTOPSY

The decision to develop our modules for ALEAPP was not arbitrary. Our primary objective was to create a tool that could be utilized independently or seamlessly into the Autopsy forensic platform (Barr-Smith et al., 2021). Autopsy, being Java-based, requires its extension modules to be written either in Java or in Jython, a Python interpreter that runs within the Java Virtual Machine (Autopsy Community, 2023).

ALEAPP fulfills both of these requirements. It is already integrated into Autopsy, ensuring that new versions of Autopsy are equipped with the latest version of ALEAPP. Therefore, modules part of ALEAPP are automatically integrated into new Autopsy releases without requiring additional developer intervention. Since our modules have been accepted and merged into ALEAPP 3.1.8, they will be readily available as soon as Autopsy is updated with ALEAPP's latest version.

5.7 THE LEAPP ECOSYSTEM

ALEAPP is not the only tool developed by Alexis Brigoni. He is developing and managing several other tools with a similar goal to ALEAPP. For example, another popular tool is ILEAPP [6] which is very similar to ALEAPP but focused on extracting data from iOS applications and Windows Logs Events And Protobuf Parser (WLEAPP) [7] that aims for forensic artifacts found in the Windows OS. Another tool is Vehicles Logs Events And Protobuf Parser (VLEAPP) [8], which

5 https://highlightjs.org
6 https://github.com/abrignoni/iLEAPP
7 https://github.com/abrignoni/WLEAPP
8 https://github.com/abrignoni/WLEAPP

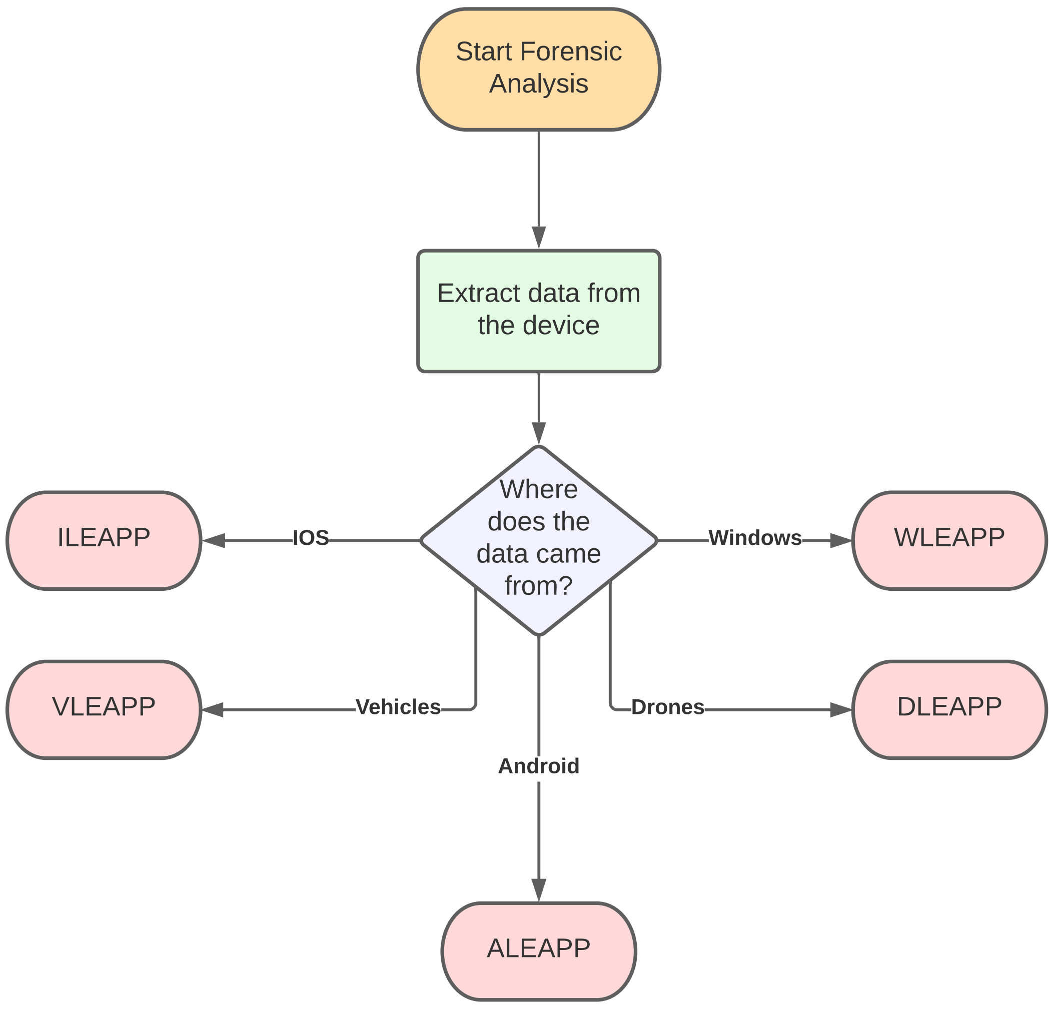


Figure 28: LEAPP Ecosystem

targets data extracted from a vehicle OS and Drones Logs Events And Protobuf Parser (DLEAPP) [9] focusing on data extracted from drones.

All applications are similar. This means that the method of adding modules and creating features is the same across all. So a developer can create a feature for ALEAPP and easily import it to ILEAPP, for example. The goal is to create an ecosystem where analysts can use the appropriate LEAPP application to analyze it and generate a report.

Figure 28 shows a diagram that shows a possible way to look at the LEAPP ecosystem.

9 https://github.com/abrignoni/DLEAPP

5.8 SUMMARY

To summarize, we successfully created 23 ALEAPP based modules related to the Garmin Connect application, extracting essential data from the private directory and the API. Using these modules, an ALEAPP user can generate a case report that he/she can then use to interpret the application's data. In this report, he/she will find various artifacts for a given timespan, such as Heart Rate, sleep time, activities realized, and GPS coordinates for a given workout, to list just a few.

In addition, we also created a set of five features to improve the report generation of ALEAPP. These features can be adapted to other modules and tools. Furthermore, since our changes were already committed to ALEAPP, they will also be present in future versions of Autopsy (ALEAPP for autopsy is currently only supported for Windows).

6 ADDITIONAL FITNESS APPLICATIONS

There are various types of fitness applications for mobile devices and wearables, as explained earlier in the introduction of this project. This project focused on creating a complete forensic analysis of the Garmin Connect application, discovering the forensic artifacts stored and transmitted in the communication. However, another popular type of fitness application is called **Running Applications**. The focus of these applications is to record runs made by the user and share them with other users, creating a type of Social Network with gamification features such as we had seen in the Garmin Connect application. These applications are generally agnostic to wearables and can communicate with companion applications and devices (for example, Garmin Connect and Fitbit). Most of these applications are related to fitness brands such as Nike and Adidas and are also a way of promoting their fitness wear. A run can be started from the application or a smartwatch if compatible. Some applications import the workouts from companion applications if the user connects them.

We used the research by Hassenfeldt et al. (2019) detailed in Chapter 2 as an initial base since we studied a group of applications already analyzed before in that research. However, we aimed to use the process done previously for Garmin Connect as a base for the analysis of these applications.

6.1 MAIN FINDINGS

After our extensive study of the Garmin Connect application, we decided to enrich this project by using the knowledge gained to analyze a group of the most popular Social Running applications. In total, we analyzed six applications then being:

- Adidas Running
- Map My Walk
- Nike Run Club
- Pumatrac
- Runkeeper

Table 28: Analyzed Applications

APPLICATION	DEVELOPER	VERSION	DOWNLOADS	VERSION RELEASE DATE
Adidas Running [1]	Adidas Runtastic	13.6	50M+	14/03/2023
Map My Walk [2]	MapMyFitness, Inc.	23.5.2	10M+	21/03/2023
Nike Run Club [3]	Nike, Inc.	4.21.0	10M+	02/03/2023
Pumatrac [4]	PUMA SE	4.19.9	1M+	18/09/2022
Runkeeper [5]	ASICS Digital, Inc.	14.3	10M+	17/03/2023
Strava [6]	Strava Inc.	299.19	50M+	22/03/2023

- Strava

Table 28 shows some public information related to the application studied using data from the Play Store. All applications have a user base in the millions and have been around for quite some time.

The analysis done is shorter compared to the one done to Garmin Connect and focuses mainly on the study of the *post-mortem* data extracted via ADB from a rooted smartphone. We started by doing the same process as in Chapter 3. Therefore, to fill the application with data, we:

1. Downloaded each application
2. Generated data with the application (creating an account, connecting with wearable, and doing activities)
3. Extract the public and private directory using ADB
4. Analyze the data in search of forensic artifacts

Table 29 briefly presents the data we managed to acquire from these applications. The methods used to analyze the applications follow the ones explained earlier in Chapter 3.

All the studied applications are relatively similar and provide roughly the same functionalities, and thus is no wonder that they hold identical data, Figures 29

1 https://play.google.com/store/apps/details?id=com.runtastic.android
2 https://play.google.com/store/apps/details?id=com.mapmywalk.android2
3 https://play.google.com/store/apps/details?id=com.nike.plusgps
4 https://play.google.com/store/apps/details?id=com.pumapumatrac
5 https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro
6 https://play.google.com/store/apps/details?id=com.strava

Table 29: Applications Analyzed

APPLICATION	ARTIFACTS	DIRECTORY	FILES
Adidas Running	Activity Data User Data	Private Directory	`db` `user.db`
Map My Walk	Activity Data User Data	Private Directory	`workout.db` `mmdk_user`
Nike Run Club	Activity Data	Private Directory	`com.nike.nrc.room`
Pumatrac	Activity Data User Data	Private Directory	`pumatrac-db`
Runkeeper	Activity Data User Data	Private Directory	`RunKeeper.sqlite` `com.fitnesskeeper.runkeeper.pro_preferences.xml`
Strava	Activity Data	Public Directory	FIT Files

and 30 show the main dashboard of the Nike Run Club application, the rest of the applications follow a very similar UI.

Furthermore, except for Strava, all applications stored data in their private directories, mainly in the SQLite database and Shared Preferences. They primarily store data related of the activities recorded by the application. There are slight variations depending on the application, but they all store data such as:

- Calories
- Distance
- Duration
- GPS Coordinates

The applications stored GPS coordinates in different ways. For example, Nike Run Club and Adidas Running store the coordinates in a polyline format, such as Garmin Connect. However, the other applications store each set of latitude and longitude as an individual record in the respective database table.

Data related to the user common to all application are as follows:

- Name
- Email
- Height/weight
- Birth date
- Profile Picture
- Country

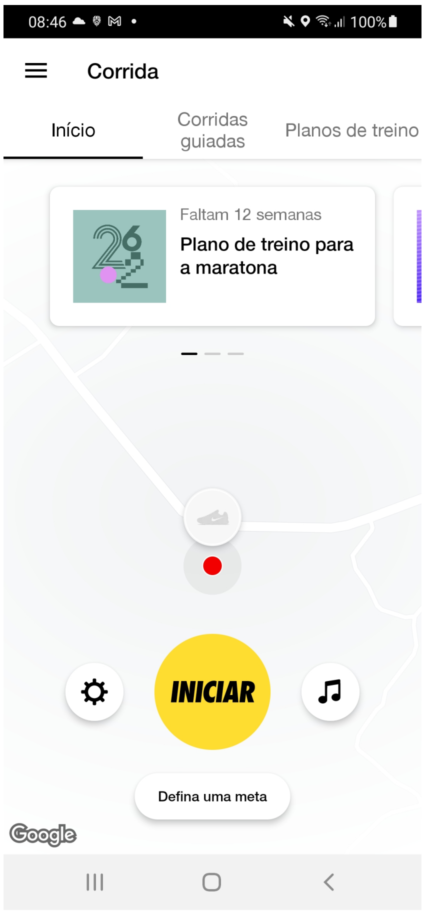


Figure 29: Main screen of Nike Run Club

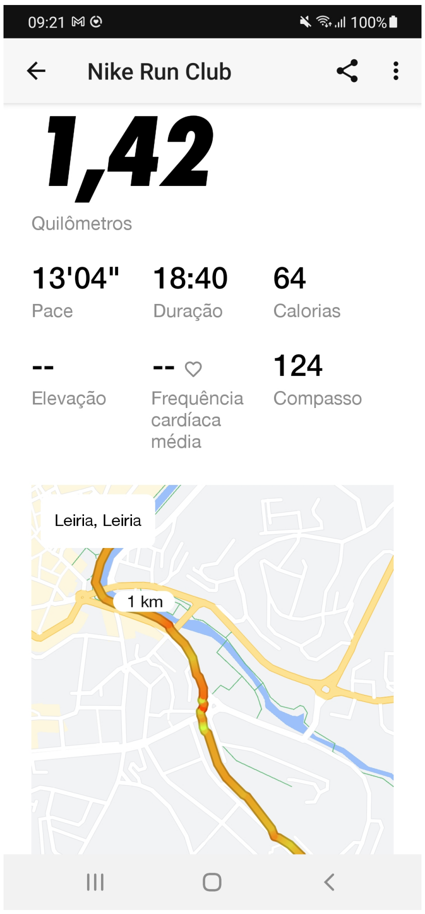


Figure 30: Nike Run Club Activity Summary

The Runkeeper application did not store user'sdata in the database. However, it was possible to obtain this data from the Shared Preferences XML files since they contain the data related to the user account information.

6.1.1 *Strava*

One application that stands out from the rest is Strava. Indeed, we did not find any data stored in the databases from Strava nor in any folder of the private directory. Like Garmin Connect, Strava purges the databases the moment it connects to the internet and uses its API to fetch data from the cloud. This fact is unsurprising since, in 2018, Strava contained a vulnerability that exposed various military bases in Israel (Gritten, 2022). Another widely known case was from Adam Jones where a burglar stole all his bicycles, the burglar analyzed Adams's Strava publication to identify where we lived and so rob is garage (Trembley, 2018). Since then, the application has been more privacy and secure-focused adding various new features such as hiding the taken route as shown in Figure 31.

However, by analyzing its public directory, we discovered that it contained FIT Files. These files were stored in a subfolder called `files` as shown in Figure 32.

In Chapter 4, we discussed what a FIT file was and how we tried to parse it. We tried the same process again and this time, we could successfully parse the data. Each FIT file contained the data related to the recorded activity, such as the timespan, distance, and GPS coordinates. This finding is of great relevance since valuable data are stored in the public directory. This means that in an investigation, the analyst can obtain this data without root privileges compared to the data from the other applications saved to the private directory. However, since the data is public, any application can access it, and an attacker can steal it. From a forensic point of view, this discovery was precious.

To ease the analysis of the FIT files we created a script very similar to our first script Polyline2GPS, this one is called FIT2GPS [7]. It decodes the given Strava FIT files to the corresponding GPS coordinates, and generates an output with the route taken by the user. The application can also generate an Excel file with the coordinates data obtained from the geopy library.

7 https://github.com/labcif/FIT2GPS

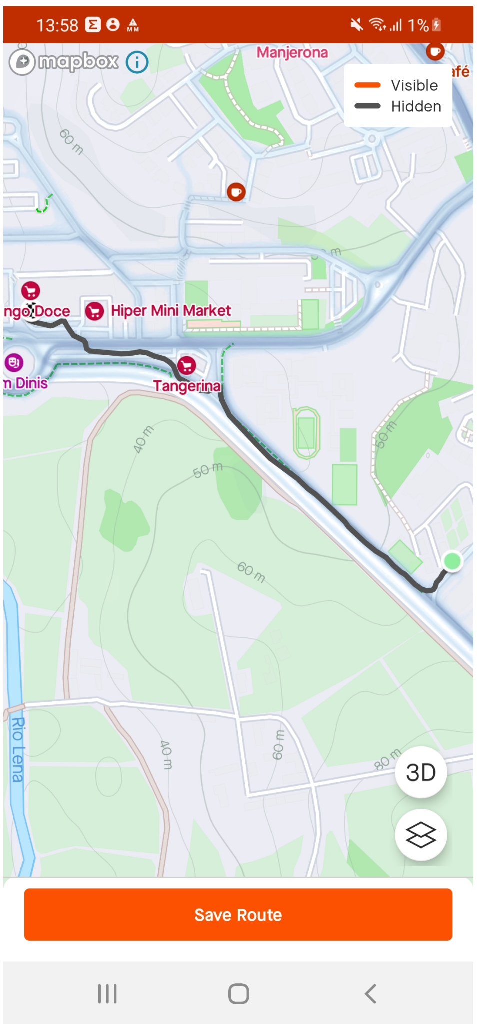


Figure 31: Strava hidden Route

```
com.strava
└── files/
```

Figure 32: Directories inside the public folder of Strava

Table 30: Applications Analyzed

MODULE	USE
`AdidasActivities.py`	Extracts the activity data from Adidas Running
`AdidasUsers.py`	Extracts the user data from Adidas Running
`MMWActivities.py`	Extracts the activity data from Map My Walk
`MMWUsers.py`	Extracts the user data from Map My Walk
`NikeActivities.py`	Extracts the activity data from Nike Run Club
`NikeAMoments.py`	Extracts the moment-to-moment information of the activity from Nike Run Club
`NikeNotifications.py`	Extracts the application notification from Nike Run Club
`NikePolyline.py`	Extracts the GPS data of the activity from Nike Run Club
`PumaActivities.py`	Extracts the activity data from Pumatrac
`PumaUsers.py`	Extracts the user data from Pumatrac
`RunkeeperActivities.py`	Extracts the activity data from Runkeeper
`RunkeeperUsers.py`	Extracts the user data from Runkeeper
`Strava.py`	Extracts the GPS data of the activity from Strava

6.1.2 *Aleapp modules*

Since all the applications were similar, the acquisition process was mostly the same, as shown in Figure 33.

Just like for the Garmin Connect application, we decided to create a series of modules for ALEAPP for each of these applications to enhance the framework further. We created in total 13 modules presented in Table 30:

6.1.3 *Summary*

Patterns could be discerned within the directories of the applications. Typically, the "public directory" mainly stored cache data for GPS tiles, which was not readable and lacked any forensic evidence. Conversely, within the "private directory," the majority of forensic artifacts were discovered in the `database` and `sharedpreferences` folders. The database typically held information regarding user activities and associated data, while shared preferences contained data linked to user accounts, as

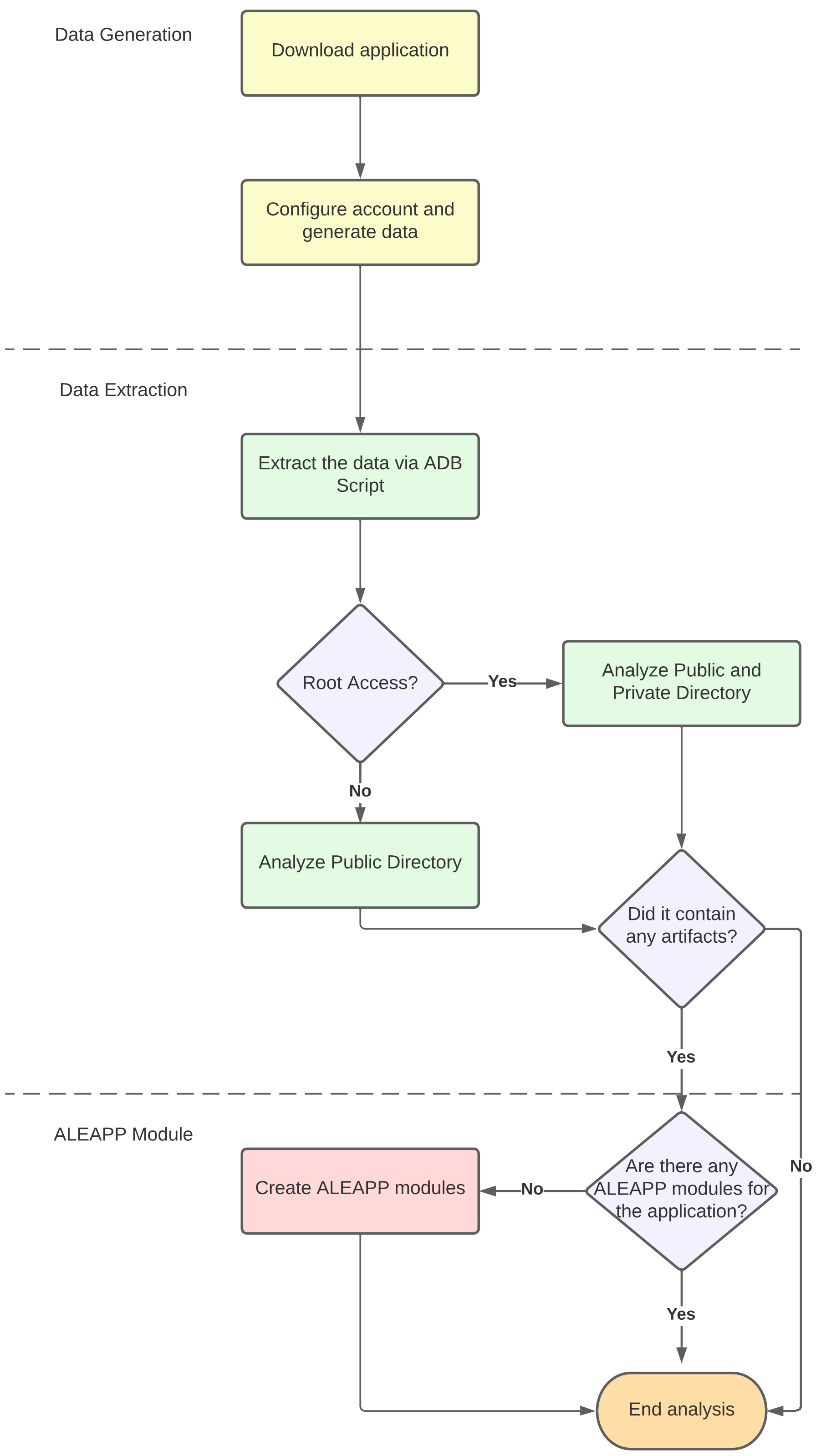

Data Generation
Download application
Configure account and generate data
Data Extraction
Extract the data via ADB Script
Root Access?
Yes
Analyze Public and Private Directory
No
Analyze Public Directory
Did it contain any artifacts?
Yes
No
ALEAPP Module
Create ALEAPP modules
No
Are there any ALEAPP modules for the application?
Yes
End analysis

Figure 33: Acquisition Process

well as a means to retrieve this data through a form in case it was not stored in the database.

In the end, gathering valuable information about these six running applications was possible. Their most prominent artifact is the GPS coordinates related to activities that can be used to locate with datetime the application's user.

7 CONCLUSION

There is no denying that wearables will continue to evolve with the amount of information stored by their respective companion applications and other fitness applications. With this fact comes the rising concern of potential privacy flaws and attacks that can occur to these kinds of applications. Furthermore, the importance it can hold in a criminal investigation as the data collected by these applications and devices can be the key to finding the culprit.

This work aimed to execute a complete forensic analysis of a specific fitness application called Garmin Connect, developed by Garmin and using the smartband Garmin Vivosmart 4 to collect data. This analysis was done on the Android version of the application using a rooted Samsung A40.

As starting point, we made a literature review in Chapter 2 of studies and forensic investigations done on wearables and companion applications. We also reviewed studies that involved the execution of dynamic analysis to other Android applications, since there is scarce documentation of this process. Lastly, studies that focused on ALEAPP its use in forensic investigations and the development of modules for it.

We started analyzing the Garmin Connect application in Chapter 3. This chapter was large and focused on performing the application's static or post-mortem analysis. We started by:

1. Gathering Information of the application
2. Prepare our acquisition environment
3. Downloading and using the application (generate data)
4. Extract and analyze the data
5. Use MobSF to find vulnerabilities

We continued our analysis in chapter 4. Here, we did a dynamic analysis of the Garmin Connect application using various offensive and reconnaissance tools to gather all the information related to the network communication and applications code to find vulnerabilities.

After completing our analysis, we created various plugins and features for the open-source ALEAPP framework, which is used in forensic analysis of Android applications to find artifacts and generate a report with the data found. In chapter 5, we detailed the plugins created and their development process.

Lastly, we combined all the work done for Garmin Connect and applied it to analyze a group of fitness applications related to running that can receive data from Garmin Connect. In Chapter 6, we analyzed the data stored (static analysis) by these applications and created, in addition, various new modules for ALEAPP.

In summary, the goals we had for this project were:

- Doing a full forensic analysis of the Garmin Connect application for Android
- Finding potential vulnerabilities in Garmin Connect
- Developing Open Source modules for ALEAPP
- Analyze other fitness applications with the knowledge gained

The forensic analysis of the Garmin Connect application helped discover a large volume of forensic artifacts and methods to obtain them. It was instrumental in creating its respective modules and, lastly, served as a pathway for analyzing other applications of similar nature.

7.1 MAIN CONTRIBUTIONS

Ultimately, this work contributed to several aspects of the Digital Forensics landscape. By doing a complete analysis of the Garmin Connect application, we improved the existing static analysis knowledge of the application made by Hutchinson et al. (2022) by analyzing more files, using different methods, finding more artifacts, and even creating some Python Scripts to improve the analysis. In addition, we made a dynamic analysis of the application. This process is rarer to find, especially for companion applications. We believe that our work is the first to perform a dynamic analysis of the application Garmin Connect. Our work can be used as a blueprint for the analysis of other similar applications by using methods and processes similar to ours. We also managed to create a script that lets the analyst potentially recover multiple types of records for any given period from the application just by having the phone connected to the machine. This analysis made it possible to use the knowledge gained and to study six other fitness applications and find all the artifacts stored by them, expanding the forensic knowledge of this type of application.

Table 31: Code Repositories

NAME	USE
ADB-Extractor[1]	Repository with the script to extract data via ADB
Polyline2GPS[2]	Repository with the script for decoding polylines to HTML or Keyhole Markup Language (KML)
FindPackage[3]	Repository with the script to find the applications package name
FIT2GPS[4]	Repository with the script for decoding FIT Files to HTML or KML
Garmin-Connect-API-Extractor[5]	Repository with the script for extracting Garmin Connect API data
Garmin-Connect-Database[6]	Repository with database diagrams and scripts related to Garmin Connect created in DBDiagram.io
GC4AA[7]	Fork from the ALEAPP repository for the modules

Our most significant contribution was developing various features and modules for the ALEAPP framework. With the analysis of the Garmin Connect application and the other six applications, we created many modules and features to enrich this framework and, in the future, to be integrated into the Autopsy framework.

All developed scripts during are open source licensed under the GPL-3 license and hosted in our organization LabCIF on Github. Table 31 shows all the code repositories created for this project.

7.2 FUTURE WORK

Even though we managed to complete all our goals, there is still room for improvement. This project could be enhanced and perfected in various ways.

In the future, we would like to improve our dynamic analysis and use the tool Frida more in-depth to alter the application and see if we can find any new vulnerabilities.

1 https://github.com/labcif/ADB-Extractor
2 https://github.com/labcif/Polyline2GPS
3 https://github.com/labcif/FindPackage
4 https://github.com/labcif/FIT2GPS
5 https://github.com/labcif/Garmin-Connect-API-Extractor
6 https://github.com/labcif/Garmin-Connect-Database
7 https://github.com/labcif/GC4AA

We would also like to perform a dynamic analysis of the other the other fitness applications tested and see if we could find any findings or vulnerabilities.

Lastly, it would also be interesting to analyze Garmin Connect for iOS and see whether it saves the same data and whether the application behaves similarly to the Android version. The end goal would be to compare the applications in different operating systems, the techniques and tools used and the artifacts and vulnerabilities found. This analysis would allow us to adapt our ALEAPP modules to ILEAPP, that is, the framework version for IOS applications.

BIBLIOGRAPHY

Almogbil, Atheer et al. (Aug. 2020). “Digital Forensic Analysis of Fitbit Wearable Technology: An Investigator’s Guide”. In: *Proceedings - 2020 7th IEEE International Conference on Cyber Security and Cloud Computing and 2020 6th IEEE International Conference on Edge Computing and Scalable Cloud, CSCloud-EdgeCom 2020*, pp. 44–49. DOI: 10.1109/CSCLOUD-EDGECOM49738.2020.00017.

Android Developers (2023). *Logcat command-line tool Android Studio Android Developers*. URL: https://developer.android.com/tools/logcat (visited on 07/06/2023).

Angie (Nov. 2016). *How NOT to examine SQLite WAL files | Sanderson Forensics*. URL: https://sqliteforensictoolkit.com/how-not-to-examine-sqlite-wal-files/ (visited on 06/09/2023).

Autopsy Community (2023). *Autopsy - Digital Forensics*. 2023-05-29. URL: https://www.autopsy.com/.

Bang, Ankur O, Udai Pratap Rao, and Amit A Bhusari (2022). “A Comprehensive Study of Security Issues and Research Challenges in Different Layers of Service-Oriented IoT Architecture”. In: *Cyber Security and Digital Forensics*, pp. 1–43.

Bangert, Bill (2022). *University of Cincinnati joins national study on stroke prevention using wearables | University Of Cincinnati*. URL: https://www.uc.edu/news/articles/2022/11/university-of-cincinnati-joins-national-study-on-stroke-prevention-using-wearables.html (visited on 11/15/2022).

Bardou, Romain et al. (2012). “Efficient padding oracle attacks on cryptographic hardware”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7417 LNCS, pp. 608–625. ISSN: 03029743. DOI: 10.1007/978-3-642-32009-5_36/COVER. URL: https://link.springer.com/chapter/10.1007/978-3-642-32009-5_36.

Barr-Smith, Frederick et al. (2021). “Dead Man’s Switch: Forensic Autopsy of the Nintendo Switch”. In: *Forensic Science International: Digital Investigation* 36, p. 301110.

Barros, António et al. (2022). “Forensic Analysis of the Bumble Dating App for Android”. In: DOI: 10.3390/forensicsci2010016. URL: https://doi.org/10.3390/forensicsci2010016.

Brignoni, Alexis (2023). *abrignoni/ALEAPP: Android Logs Events And Protobuf Parser*. 2023-05-29. URL: https://github.com/abrignoni/ALEAPP.

Byambasuren, Oyungerel, Elaine Beller, and Paul Glasziou (June 2019). “Current Knowledge and Adoption of Mobile Health Apps Among Australian General Practitioners: Survey Study”. In: *JMIR Mhealth Uhealth* 7 (6). ISSN: 22915222. DOI: 10.2196/13199. URL: https://mhealth.jmir.org/2019/6/e13199.

Cheng, Chris Chao Chun et al. (July 2021). “LogExtractor: Extracting digital evidence from android log messages via string and taint analysis”. In: *Forensic Science International: Digital Investigation* 37. ISSN: 26662817. DOI: 10.1016/J.FSIDI.2021.301193.

Dawson, Liam and Alex Akinbi (July 2021). “Challenges and opportunities for wearable IoT forensics: TomTom Spark 3 as a case study”. In: *Forensic Science International: Reports* 3. ISSN: 26659107. DOI: 10.1016/J.FSIR.2021.100198.

Delija, D. et al. (2022). “How to do a forensic analysis of Android 11 artifacts”. In: *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology, MIPRO 2022 - Proceedings*, pp. 1042–1047. DOI: 10.23919/MIPRO55190.2022.9803540.

Developers, Android (2023). *Android 8.0 Behavior Changes - Android Developers*. URL: https://developer.android.com/about/versions/oreo/android-8.0-changes#security-all (visited on 07/06/2023).

Domingues, Patrício, José Francisco, and Miguel Frade (June 2023). “Post-mortem digital forensics analysis of the Zepp Life android application”. In: *Forensic Science International: Digital Investigation* 45, p. 301555. ISSN: 2666-2817. DOI: 10.1016/J.FSIDI.2023.301555.

Fereidooni, Hossein et al. (Aug. 2017). “Fitness Trackers: Fit for Health but Unfit for Security and Privacy”. In: *Proceedings - 2017 IEEE 2nd International Conference on Connected Health: Applications, Systems and Engineering Technologies, CHASE 2017*, pp. 19–24. DOI: 10.1109/CHASE.2017.54.

Gaffney, Kevin P. et al. (2022). “SQLite: Past, Present, and Future”. In: *Proceedings of the VLDB Endowment* 15 (12), pp. 3535–3547. ISSN: 21508097. DOI: 10.14778/3554821.3554842.

Garmin Ltd. (2023). *FIT Protocol | FIT SDK | Garmin Developers*. 2023-05-27. URL: https://developer.garmin.com/fit/protocol/.

Garmin Portugal (2023). URL: https://www.garmin.com/pt-PT/ (visited on 07/06/2023).

Gritten, David (June 2022). *Strava app flaw revealed runs of Israeli officials at secret bases - BBC News*. URL: https://www.bbc.com/news/world-middle-east-61879383 (visited on 04/02/2023).

Hassenfeldt, Courtney et al. (Aug. 2019). "Map My Murder! A digital forensic study of mobile health and fitness applications". In: *ACM International Conference Proceeding Series*. DOI: 10.1145/3339252.3340515.

Hildenbrand, Jerry (2023). *Best Android phones for rooting and modding 2023 | Android Central*. URL: https://www.androidcentral.com/best-phone-rooting-and-modding (visited on 07/06/2023).

Hutchinson, Shinelle et al. (Sept. 2022). "Investigating Wearable Fitness Applications: Data Privacy and Digital Forensics Analysis on Android". In: *Applied Sciences 2022, Vol. 12, Page 9747* 12 (19), p. 9747. ISSN: 2076-3417. DOI: 10.3390/APP12199747. URL: https://www.mdpi.com/2076-3417/12/19/9747/htm%20https://www.mdpi.com/2076-3417/12/19/9747.

Kang, Serim, Soram Kim, and Jongsung Kim (Mar. 2020). "Forensic analysis for IoT fitness trackers and its application". In: *Peer-to-Peer Networking and Applications* 13 (2), pp. 564–573. ISSN: 19366450. DOI: 10.1007/S12083-018-0708-3/TABLES/6. URL: https://link.springer.com/article/10.1007/s12083-018-0708-3.

Kent, Karen et al. (2006). "Special Publication 800-86 Guide to Integrating Forensic Techniques into Incident Response Recommendations of the National Institute of Standards and Technology". In.

Kim, Minju et al. (July 2022). "Digital forensic analysis of intelligent and smart IoT devices". In: *Journal of Supercomputing*, pp. 1–25. ISSN: 15730484. DOI: 10.1007/S11227-022-04639-5/TABLES/7. URL: https://link.springer.com/article/10.1007/s11227-022-04639-5.

Lamalva, Grace and Suzanna Schmeelk (2020). "MobSF: Mobile Health Care Android Applications Through the Lens of Open Source Static Analysis". In: *2020 IEEE MIT Undergraduate Research Technology Conference, URTC 2020*. DOI: 10.1109/URTC51696.2020.9668870.

Li, Xiang et al. (Dec. 2016). "An Android malware detection method based on AndroidManifest file". In: *Proceedings of 2016 4th IEEE International Conference on Cloud Computing and Intelligence Systems, CCIS 2016*, pp. 239–243. DOI: 10.1109/CCIS.2016.7790261.

Lovejoy, Ben (2022). *Smartwatch Market Size, Share| 2022 - 27 | Industry Report*. 2023-02-26. URL: https://www.mordorintelligence.com/industry-reports/smartwatch-market.

MacDermott, Aine et al. (June 2019). "Forensic analysis of wearable devices: Fitbit, Garmin and HETP Watches". In: *2019 10th IFIP International Conference on New Technologies, Mobility and Security, NTMS 2019 - Proceedings and Workshop*. DOI: 10.1109/NTMS.2019.8763834.

Miller, Preston and Chapin Bryce (2016). *Learning Python for forensics : leverage the power of Python in forensic investigations*. ISBN: 9781789341690. URL: https://www.packtpub.com/product/learning-python-for-forensics-second-edition/9781789341690.

Mirza, Mohammad Meraj, Akif Ozer, and Umit Karabiyik (Nov. 2022). "Mobile Cyber Forensic Investigations of Web3 Wallets on Android and iOS". In: *Applied Sciences 2022, Vol. 12, Page 11180* 12 (21), p. 11180. ISSN: 2076-3417. DOI: 10.3390/APP122111180. URL: https://www.mdpi.com/2076-3417/12/21/11180/htm%20https://www.mdpi.com/2076-3417/12/21/11180.

Mueller, Bernhard et al. (2022). *OWASP MASTG - OWASP Mobile Application Security*. URL: https://mas.owasp.org/MASTG/.

okta (2023). *Differences Between OAuth 1 and 2 - OAuth 2.0 Simplified*. URL: https://www.oauth.com/oauth2-servers/differences-between-oauth-1-2/ (visited on 07/06/2023).

Ramírez-López, Francisco José et al. (Nov. 2019). "A Framework to Secure the Development and Auditing of SSL Pinning in Mobile Applications: The Case of Android Devices". In: *Entropy 2019, Vol. 21, Page 1136* 21 (12), p. 1136. ISSN: 1099-4300. DOI: 10.3390/E21121136. URL: https://www.mdpi.com/1099-4300/21/12/1136/htm%20https://www.mdpi.com/1099-4300/21/12/1136.

Razaghpanah, Abbas et al. (Feb. 2018). "Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem". In: DOI: 10.14722/NDSS.2018.23353.

Sachdeva, Shefali, R. Jolivot, and Worawat Choensawat (2018). "Android Malware Classification based on Mobile Security Framework". In.

Saha, Renata, Sayan Sarkar, and Soumya Kanti Datta (Oct. 2017). "Balancing security & sharing of fitness trackers' data". In: *2017 1st International Conference on Electronics, Materials Engineering and Nano-Technology, IEMENTech 2017*. DOI: 10.1109/IEMENTECH.2017.8076942.

Schiefer, Michael (2015). "Internet of Things Security Evaluation of nine Fitness Trackers". In: URL: www.av-test.org.

Shahriar, Hossain et al. (2019). *An Exploratory Analysis of Mobile Security Tools COMPSAC Message View project A Signature-Based Intrusion Detection System for Web Applications based on Genetic Algorithm View project*. URL: https://digitalcommons.kennesaw.edu/ccerp/2019/research/4.

Shweta, Ganjoo (2019). *GPS data from Garmin smartwatch helps police catch a man convicted of two murders.* URL: https://www.indiatoday.in/technology/news/story/how-a-garmin-smartwatch-helped-police-catch-a-man-convicted-of-two-murders-1435570-2019-01-21.

similarweb (2023). *Garmin Connect™ App Stats: Downloads, Users and Ranking in Google Play | Similarweb.* URL: https://www.similarweb.com/app/google-play/com.garmin.android.apps.connectmobile/statistics/ (visited on 07/15/2023).

Skulkin, Oleg., Donnie. Tindall, and Rohit. Tamma (2018). *Learning Android Forensics : Analyze Android Devices with the Latest Forensic Tools and Techniques, 2nd Edition.* Packt Publishing Ltd, p. 324. ISBN: 9781789131017.

SQLite Community (Feb. 2023). *Pragma statements supported by SQLite.* 2023-06-16. URL: https://www.sqlite.org/pragma.html#pragma_auto_vacuum.

Tamma, Tamma|Oleg Skulkin Rohit (2020). *Practical Mobile Forensics Forensically investigate and analyze iOS, Android, and Windows 10 devices, 4th Edition.* Packt Publishing. ISBN: 9781838647520.

Tangari, Gioacchino et al. (Sept. 2021). "Analyzing security issues of android mobile health and medical applications". In: *Journal of the American Medical Informatics Association* 28 (10), pp. 2074–2084. ISSN: 1527974X. DOI: 10.1093/JAMIA/OCAB131. URL: https://academic.oup.com/jamia/article/28/10/2074/6335525.

Technologies, Positive (2019). *Mobile Application Security Threats and Vulnerabilities 2019: Mobile Device Security - Attacks Research.* URL: https://www.ptsecurity.com/ww-en/analytics/mobile-application-security-threats-and-vulnerabilities-2019/ (visited on 11/15/2022).

Trembley, Philippe (Sept. 2018). *Thieves allegedly use Strava to identify and steal cyclist's $21,000 bike collection - Canadian Cycling Magazine.* URL: https://cyclingmagazine.ca/sections/news/thieves-allegedly-use-strava-to-help-steal-cyclists-21000-bike-collection/ (visited on 07/14/2023).

Unuchek, Roman (June 2017). *Android: To root or not to root | Kaspersky official blog.* URL: https://www.kaspersky.com/blog/android-root-faq/17135/ (visited on 07/06/2023).

Vasilaras, Alexandros et al. (Sept. 2022). "Retrieving deleted records from Telegram". In: *Forensic Science International: Digital Investigation* 43, p. 301447. ISSN: 2666-2817. DOI: 10.1016/J.FSIDI.2022.301447.

Watts, Amanda (2017). *Police use murdered woman's Fitbit movements to charge her husband - CNN.* URL: https://edition.cnn.com/2017/04/25/us/fitbit-womans-death-investigation-trnd/index.html (visited on 11/04/2022).

Williams, Joseph et al. (May 2021). “Forensic Analysis of Fitbit Versa: Android vs iOS”. In: *Proceedings - 2021 IEEE Symposium on Security and Privacy Workshops, SPW 2021*, pp. 318–326. DOI: 10.1109/SPW53761.2021.00052.

Xu, Hui et al. (Dec. 2020). “Layered obfuscation: a taxonomy of software obfuscation techniques for layered security”. In: *Cybersecurity* 3 (1), pp. 1–18. ISSN: 25233246. DOI: 10.1186/S42400-020-00049-3/FIGURES/10. URL: https://cybersecurity.springeropen.com/articles/10.1186/s42400-020-00049-3.

Yoon, Yung Han and Umit Karabiyik (Sept. 2020). “Forensic Analysis of Fitbit Versa 2 Data on Android”. In: *Electronics 2020, Vol. 9, Page 1431* 9 (9), p. 1431. ISSN: 2079-9292. DOI: 10.3390/ELECTRONICS9091431. URL: https://www.mdpi.com/2079-9292/9/9/1431/htm%20https://www.mdpi.com/2079-9292/9/9/1431.

Zhu, Ningxian (Apr. 2021). “Security of CORS on LocalStorage”. In: *Proceedings - 2021 International Conference on Internet, Education and Information Technology, IEIT 2021*, pp. 141–146. DOI: 10.1109/IEIT53597.2021.00038.

APENDIXES

A MOBSF SETUP

MobSF is a Python application which relies on the Django Framework. For this analysis, we used version 3.6.2. Being a Python application, running it in different operating systems without issue is possible, but the configuration can be different. To run MobSF, the user needs to install the dependencies shown in Table 32 based on their OS.

In case the user is using macOS, he/she also needs to execute the following commands inside the Python folder:

```
Update Shell Profile. command
Install Certificates. command
```

After having all the prior dependencies installed, we can clone the project from GitHub:

```
git clone https://github.com/MobSF/Mobile-Security-Framework-MobSF.git
cd Mobile-Security-Framework-MobSF
```

Next one needs to, run the installation script. We run the setup shell script in the POSIX systems. In Windows, we run the Batch File (BAT) script.

```
./setup.sh
or
setup.bat
```

Since MobSF is made in Django, the interaction with the tool is done via a web interface whose webserver runs on localhost. For POSIX systems, we execute the **run.sh** script and specify the IP and port where we want to run MobSF. For Windows, the process is the same through the BAT file. However, we use the bat file.

```
./run.sh 127.0.0.1:8000
or
run.bat 127.0.0.1:8000
```

Table 32: MobSF dependencies

TOOL	OPERATING SYSTEM
Git	All
JDK 8 or above	All
Microsoft Visual C++ Build tools	Windows
OpenSSL non-light	Windows
Python 3.8 or above	All
wkhtmltopdf	All
XCode Command line Tools	macOS

After that, we can access MobSF in our browser on the specified IP and port. To start the static analysis, we need to provide the APK file from the application. We used the script made in the acquisition phase to extract the application file from the smartphone. An alternative could be downloading the application from an APK store like Aptoid or APKPure. After uploading the file, MobSF starts the analysis process. The time this process takes depends on the size and complexity of the application. In our case, this process took nearly four hours to complete.

DECLARATION

I declare, under oath, that the work presented in this Project, with the title *"Forensic Analysis of the Garmin Connect Android Application "*, is original and is made by Fabian Pereira Nunes (2210511) under the guidance of Professor Miguel Monteiro de Sousa Frade, Ph.D. and Professor Patrício Rodrigues Domingues, Ph.D..

Leiria, July of 2023

Assinado por: **FABIAN PEREIRA NUNES**
Num. de Identificação: BI15924410
Data: 2023.07.22 12.39.09 GMT Daylight time

CHAVE MÓVEL

Fabian Pereira Nunes