POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

AutomIoT: Generating Smart Devices Interactions Using Android UI Automator for IoT Forensics

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Angelo Claudio Re**

Student ID: 894001
Advisor: Prof. Alessandro Cesare Enrico Redondi
Co-advisors: Dott. Fabio Palmese
Academic Year: 2022-23

ACKNOWLEDGEMENTS

Con questo elaborato posso considerare concluso il mio percorso universitario.
È stata un'esperienza formativa incredibile, piena di soddisfazioni e difficoltà da superare. Questa tesi per me è un punto di svolta nella mia vita professionale che è iniziata ormai da un bel po'.
E' l'inizio di una nuova vita piena di nuove sfide. Sono estramente fiero di questo traguardo raggiunto con tanto sudore e fatica. Scrivere questa tesi è stato per me una sfida contro i miei limiti, che sono ancora molti.
Non so cosa mi riserverà il futuro, ma sicuramente con le qualità che ho acquisito e le cose che ho imparato al Politecnico di Milano, gli strumenti per vivere appieno la mia professione ci sono tutti.
Volevo ringraziare con tutto me stesso il Dott. Fabio Palmese, persona straordinaria in quanto a umanità, empatia, dedizione, professionalità e disponibilità. Mi hai accompagnato passo dopo passo al compimento di questo traguardo e mi hai dato tanto.
Un sincero ringraziamento va anche al mio relatore, il prof. Redondi, che ho conosciuto per la prima volta durante la triennale a Como nel corso di Fondamenti di Telecomunicazioni, e che ho avuto ancora il piacere di avere come docente nel corso Wireless Networks durante la magistrale.
Volevo ringraziare tutti i ragazzi del Antlab che ogni giorno sono il cuore pulsante di tale laboratorio, e della ricerca in ambito communication networks. L'unico rammarico che ho è quello di non avervi potuto conoscere di più e aver avuto modo di vivere appieno l'esperienza con voi in laboratorio causa impegni lavorativi, ma per il poco che ho vissuto, siete state delle persone splendide e mi avete fatto sentire a casa nonostante la mia timidezza e i miei limiti caratteriali.
Passiamo ora alla mia ragazza Aurora che ormai mi sopporta da quasi 3 anni e mezzo, grazie amore per essermi stata vicino in questo periodo piuttosto stressante.
Ringrazio i miei genitori per il sostegno e la motivazione che continuano a darmi giorno dopo giorno. Ringrazio mio fratello. I miei nonni che purtroppo non hanno potuto vedermi conseguire tale traguardo ma so che da lassù mi guardano dovunque essi siano.
Ringranzio mia Zia per la positività che mi ha trasmesso, e nel ricordarmi di non arren-

dermi mai di fronte alle avversità.
Ringrazio tutti gli amici che ho conosciuto anche grazie al Politecnico, e che mi hanno arricchito come persona (Jack, Lorenzo, Lara, Laura, Nicole, Federico, Giorgia, Federica).
Ringrazio Gabriele, Davide, Marco F., Marco S. e Giovanni con cui ho condiviso parte degli studi.
Infine ringrazio il Politecnico di Milano per avermi dato la possibilità di diventare un ingegnere, sogno che ho sempre voluto realizzare fin da bambino, quando fantasticavo sulla possibilità di inventare una macchina del tempo, o sognavo di creare un dispositivo innovativo per qualche azienda tecnologica.
Ringrazio i prof. Maggi, Zanero, Santambrogio, Capone, Matteucci, Restelli, Amigoni, Gatti, Cugola, Pradella, Svelto, Marcon, Cesana, Bestagini, e la prof. Schiaffonati per le conoscenze che mi hanno trasmesso.
In particolare il prof. Mottola del NESLab, e Andrea Maioli, per la loro disponibilità, gli insegnamenti, le lezioni, e opportunità che mi hanno dato, e la Rettrice Sciuto, una guida e un punto di riferimento da prendere come esempio.
Concludo i miei ringraziamenti con un aneddoto.
Ricordo ancora quando verso la fine della triennale non sapevo cosa sarebbe stato del mio futuro, cosa volevo, e un giorno mentre passavo in un corridoio del Politecnico in Piazza Piola, mi fermai per qualche secondo ad ammirare l'architettura della stanza in cui mi trovavo. In quel preciso istante capii cosa volevo, provai una pura sensazione di benessere e sicurezza, e decisi di andare avanti e proseguire gli studi. Il Politecnico per me rappresenta molto di più di una istituzione, o un'università, ma un posto quasi magico, che quando ho bisogno di ritrovare me stesso, oppure ho bisogno di schiarirmi le idee sarà sempre li ad accogliermi tra le sue braccia, insieme alla sua storia e a tutti i giganti che ci hanno preceduto sulle cui spalle giacciamo adesso noi.

Abstract

The field of IoT forensics has emerged in response to the increasing prevalence of IoT devices in critical infrastructure and various sectors like healthcare, smart homes, smart cities, industrial applications, and agriculture. As these devices become ubiquitous, they are potential witnesses and targets in both physical and digital crimes. Our focus lies in IoT forensics, specifically in analyzing network traffic from these devices.
A significant challenge in IoT network traffic analysis is the scarcity of labeled datasets containing both network traffic and device activity. Manual creation of such datasets is labor-intensive and time-consuming. In response, our work introduces AutomIoT, a framework designed to automate interactions between a smart device and its companion app. Leveraging the UI Automator framework, AutomIoT facilitates the creation of automated interactions and the scheduling of periodic operations for collecting network traffic data from IoT devices without human intervention. These operations are logged, enabling us to label and utilize the resulting dataset for forensic purposes
As a matter of fact, it served as a foundational element for our subsequent IoT forensics analysis, with a focus on identifying spontaneous traffic within the dataset and distinguishing it from "EVENTs" traffic, for data labeling purposes. Our analysis includes the examination of both spontaneous and triggered traffic, supported by data visualization techniques such as Time Series and Sankey Diagrams. Notably, about 60% of the dataset comprises spontaneous traffic, a key finding that will be presented and discussed.

Keywords: IoT Forensics, Internet of Things, Network Traffic Analysis, Network dataset, Mobile App Testing, Automation Testing, Android UI Automator

Sommario

Il campo della IoT Forensics è emerso in risposta alla crescente diffusione dei dispositivi IoT nelle infrastrutture critiche e in vari settori come la sanità, le smart home, le smart city, le applicazioni industriali e l'agricoltura. Man mano che questi dispositivi diventano ubiqui, rappresentano potenziali testimoni e bersagli in crimini sia fisici che digitali. Il nostro focus è sulla Iot Forensics, nello specifico sull'analisi del traffico di rete proveniente da questi dispositivi.
Una sfida significativa nell'analisi del traffico di rete IoT è la scarsità di set di dati "etichettati" contenenti sia il traffico di rete che l'attività del dispositivo. La creazione manuale di tali set di dati è laboriosa e richiede molto tempo. In risposta, il nostro lavoro introduce AutomIoT, un framework progettato per automatizzare le interazioni tra un dispositivo intelligente e la relativa app di accompagnamento. Sfruttando il framework UI Automator, AutomIoT facilita la creazione di interazioni automatizzate e la pianificazione di operazioni periodiche per la raccolta di dati di traffico di rete da dispositivi IoT senza intervento umano. Queste operazioni vengono registrate, consentendoci di etichettare e utilizzare il set di dati risultante a fini forensi.
Infatti tale set di dati è servito come elemento fondamentale per la nostra successiva analisi forense, con un focus sull'identificazione del traffico spontaneo all'interno del set di dati e sulla distinzione dal traffico degli "EVENTs", a scopo di etichettatura i dati di rete. La nostra analisi comprende l'esame sia del traffico "spontaneo" che di quello degli "EVENTs", supportato da tecniche di visualizzazione dati come i diagrammi a serie temporali e i diagrammi di Sankey. In particolare, circa il 60% del set di dati è costituito da traffico spontaneo, una scoperta chiave che sarà presentata e discussa.

Parole chiave: IoT Forensics, Internet of Things, Network Traffic Analysis, Network dataset, Mobile App Testing, Automation Testing, Android UI Automator

Contents

List of Figures

1 | Introduction

The pervasive digitization and rise of digital domains driven by technological advancements pose significant challenges in IoT forensics. The increasing number of computing devices and diverse applications in critical infrastructure intensifies the complexities digital investigators face. While the proliferation of IoT devices offers numerous benefits, it concurrently presents formidable security and forensics hurdles.

As highlighted in the IoT Analytics annual report [35], [36], [37], the IoT market in 2022 witnessed approximately 14.4 billion connected devices, with an enterprise spending of around $202 billion. Projections estimate a growth of 29 billion devices, generating $500 billion in revenues by 2027. Figures 1.1 and 1.2 illustrate this upward trend.

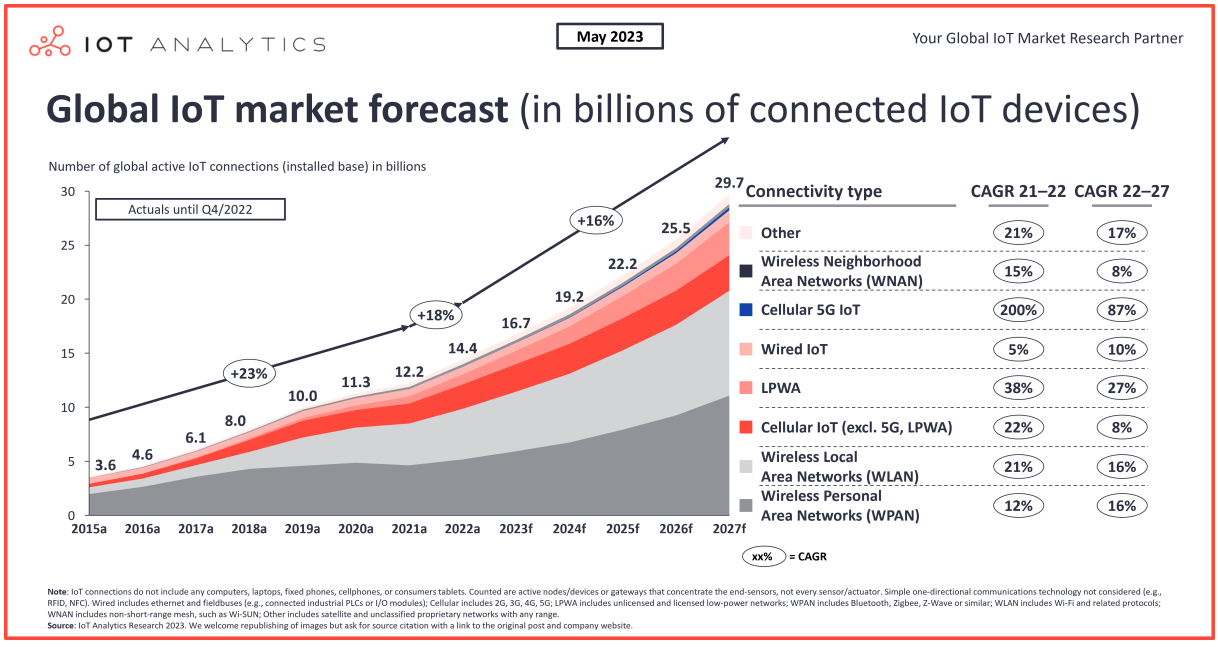

Figure 1.1: Global IoT Number of Devices Forecast

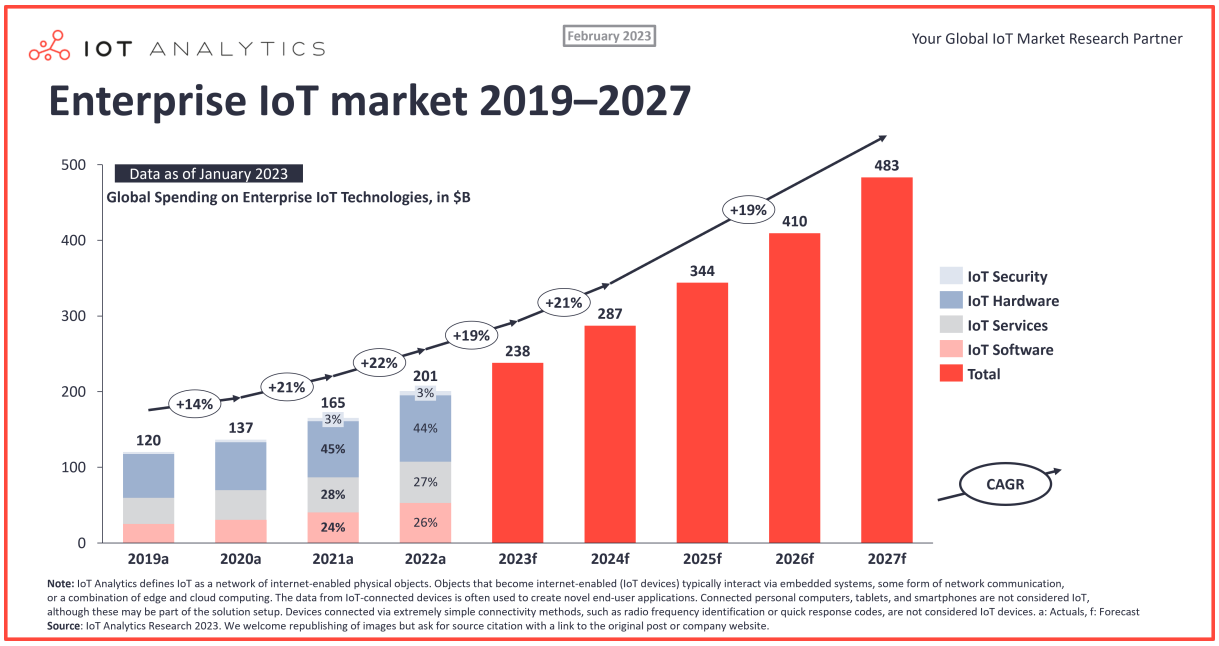

Figure 1.2: Global IoT Revenues Forecast

The huge increase of applications of IoT devices in daily human life is introducing a new witness of human daily activities: the IoT device itself. For this reason, a new forensic science, under the name of IoT Forensics, has become popular to extract useful information from IoT devices to be used to solve legal processes. The intricate landscape of IoT forensics involves a meticulous four-stage process. As the first step, investigators procure IoT devices in the **Evidence Acquisition** phase, extracting volatile and non-volatile data. During this critical step, data is stored in forensically sound and legally acceptable ways, acknowledging the diverse nature inherent in various IoT devices. Moving to the **Evidence Examination** stage, a comprehensive analysis of the acquired data unfolds. Forensic specialists deploy specialized tools to delve into the intricacies of the data, uncovering crucial insights. The **Evidence Analysis** phase commences with a trove of analyzed data. Here, investigators scrutinize the information to pinpoint the root cause of the crime, synthesizing knowledge from the earlier stages. The conclusive step, **Reporting**, marks the endpoint of the IoT forensics process. Findings and analyses are presented to the court, concluding the investigative journey.

Embarking on IoT forensics is a nuanced endeavor fraught with multifaceted challenges. The diverse array of IoT devices necessitates tailored solutions, given their distinct hardware, software, and communication protocols. The escalating complexity of IoT systems, often involving thousands of devices, introduces intricacies in the acquisition and examination processes. The heterogeneous nature of IoT devices, utilizing various communication

protocols like WiFi, Zigbee, and Z-Wave, poses an ongoing challenge for forensic investigations. Additionally, the resource constraints inherent in IoT devices—limited storage, processing, and communication capabilities—demand careful consideration in the forensic process. The dynamic nature of IoT systems, characterized by devices joining or leaving the system at any time, coupled with diverse data formats, introduces complexities in forensic processes.

In IoT forensics, investigations unfold across three critical levels. **Device Forensics** emphasizes physical devices as the primary source of evidence. Moving to **Network Forensics**, the focus shifts to communication networks, identifying potential sources of attacks across various network types. The final level, **Cloud Forensics**, underscores the pivotal role of Cloud computing in processing and storing data from resource-constrained IoT devices. Figure 1.3 shows the aforementioned levels.

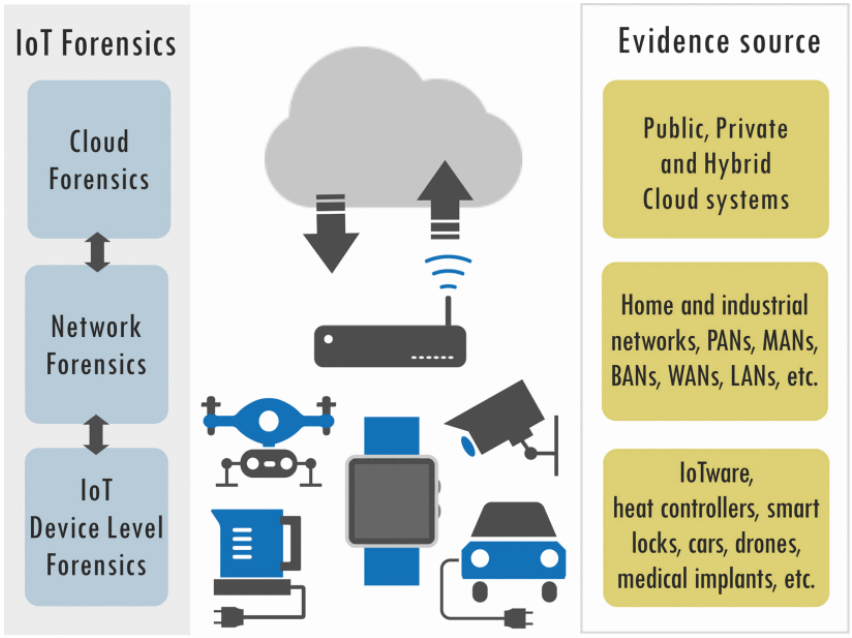

Figure 1.3: Components of IoT Forensics [77]

The ubiquity of embedded devices, known as Internet of Things (IoT) devices, has become increasingly prevalent in private, industrial, and corporate settings. Smart Home systems, particularly popular in private domains, encompass an array of devices such as sensors, security cameras, door controls, and audio systems. In the industrial sector, IoT devices like sensors and actuators are employed in manufacturing plants to optimize productivity. As these devices permeate daily life, they accumulate sensitive and private information about their surroundings. This abundance poses significant challenges for digital investigators, emphasizing the need for forensically sound and timely investigation

processes.

IoT forensics should not be confused with IoT security, which focuses on protecting devices from external attacks, and prevents the IoT devices from becoming attackers. In a specific instance outlined in [12], the IoT Device L530E [19], utilized in our thesis, exhibited four vulnerabilities—two of High severity and two of Medium severity based on the CVSS v3.1 scoring system. These vulnerabilities pertained to authentication and confidentiality, revealing inadequacies in the implemented cryptographic measures. Exploiting these weaknesses allowed an attacker not only to uncover the victim's Wi-Fi password but also to manipulate all devices within the Tapo family associated with the user's Tapo account.

The significance of gathering network data related to IoT devices extends beyond identifying vulnerabilities; it plays a crucial role in analyzing network traffic using IoT Forensics and security techniques. This data not only aids in discovering possible vulnerabilities but can also serve as valuable evidence in investigations. Our research focused on IoT smart home devices, commonly equipped with companion apps for various purposes, such as monitoring resource consumption, device scheduling patterns, or state changes (e.g., switching on/off a bulb).

One of the main problems with IoT network traffic analysis is the lack of labeled datasets, which contain both the network traffic and the device activity. Our contribution introduces a new instrument, **AutomIoT**, capable of automating the collection of a network dataset, without direct human interaction. Built upon the UI Automator, a Mobile App Testing framework for automatic Instrumented Tests, AutomIoT replicates human gestures on a mobile device's screen. The network packets produced by the interaction above have been collected using a packet analyzer tool. To support the data analysis, AutomIoT produces an activity log to label the network packets correctly with the corresponding event triggering in the device.

Finally, by using 2 data visualization techniques, namely the Time series and Sankey diagram, we have conducted a network data analysis to discern the **EVENTs** traffic, associated with the device activity, and the *spontaneous* traffic, associated with network packets generated when the mobile app and the smart device are not interacting, representing the device in a specific state without changes triggered by the controlling mobile app. Time series, along with the activity log, allowed us to label and discern the two sources of network traffic, while Sankey diagrams, let us give a quantitative measure of the *spontaneous* traffic phenomenon.

1.1. Structure of the work

The remainder of this thesis is organized as follows:

Chapter 2 comments on the latest research works in the literature concerning the main topics around IoT Forensics and Automation of Mobile App Testing, which we will use as our baseline.

Chapter 3 exposes the problem that has been faced, the solution that we devised to solve it, some theoretical backgrounds concerning mobile app testing, and the project workflow.

Chapter 4 describes the architecture of our system, the hardware and software used to build our tool, and the logic of functioning of the automated test that we developed.

Chapter 5 presents the experiments that we conduct through the use of our tool and comments on the obtained results in terms of network traffic analysis.

Chapter 6 finally concludes the work summarizing the research and reasoning on possible future works.

2 | State of the Art

In this part of the thesis, we delve into key research areas related to IoT Forensics and mobile app testing. Our focus is on understanding the methodologies for collecting forensic evidence, particularly in network data. The chapter unfolds by examining the advancements in mobile app testing, including testing methods and automation tools. The synthesis of these two areas lays the foundation for our proposed solution.

Our literature review process involved searching for relevant publications using keywords such as *"IoT Forensics"*, *"Mobile App Testing"*, *"Automation Mobile Testing"*, *"Android"*, *"IoT"*, *"Forensics"*, *"Smart Device"*, *"Monkey"*, *"Monkeyrunner"*, *"UIAutomator"*, *"Espresso"*, *"Appium"*, and *"Smart Device"*. The review encompassed leading platforms like IEEE Explore [34], ACM Digital Library [10], ScienceDirect [21], arXiv [9].

2.1. IoT Forensics

In this section, we present significant research in the field of IoT Forensics, with a primary focus on collecting evidence in the form of network traces. Notably, [39] outlines a comprehensive approach to extracting evidence from a Home Assistant Amazon Echo device across various dimensions of IoT Forensics, including device, network, and cloud levels.

Another noteworthy study [14] emphasizes the collection of labeled network datasets associated with smart devices, enabling the identification of unexpected or surreptitious device activities. The subsequent works concentrate on identifying and detecting IoT devices by analyzing network traffic from different observation points [80], [74], [73], emphasizing privacy, security, and threat reduction.

2.2. Mobile App Testing

This section delves into key research on mobile app automation testing tools, followed by an exploration of machine learning-based testing tools.

The initial study [40], despite being seven years old, remains relevant in illustrating industrial use cases for testing Android applications up to 2016. It introduced six tools: *AndroidRipper*, *EMMA*, *Monkey*, *RERAN*, *Robotium*, *Roboeletric* and *Sikuli*. Apart from EMMA, which is a toolkit for measuring and reporting Java coverage, the other ones are testing automation tools. RERAN, a Capture and Replay tool, automatically converts user events into test scripts for replaying actions. AndroidRipper stands out, employing a user-interface-driven ripper to explore an app's GUI systematically. It exposes unknown faults in code and outperforms random approaches.

AndroidRipper [2] becomes a central tool in a broader GUI testing toolset for Android applications [1]. Another tool, *iMPAcT* [49], integrates exploration, reverse engineering, and testing. It fires UI events, checks UI patterns through reverse engineering, and runs corresponding testing strategies. MobiGUITAR [3], created by the authors of AndroidRipper, automates GUI-driven testing, generating test cases based on GUI widget state abstraction.

Two noteworthy tools utilizing the Record and Replay method are based on the Robotium framework [44] and Espresso Test Recorder [50]. The former captures user events and converts them into Robotium test scripts, allowing the replay of recorded actions with the insertion of assertions for verifying Android UI component outputs. The latter, Espresso Test Recorder, employs a unique technique based on the Java Debugger to reliably capture UI interactions, generating robust, well-formed instrumentation tests applicable across various devices.

A benchmark study [84] involving 68 widely used apps suggests combining different test generation tools for improved performance. The benchmark focused on code coverage and fault-detection ability. Results revealed that the Monkey framework, a state-of-the-practice tool, demonstrated the highest method coverage in 22 of 41 apps with available data, and it also achieved the highest activity coverage in 35 apps. Additionally, Stoat, recognized as a state-of-the-art tool, triggered the highest number of unique crashes in 23 apps. The experimental outcomes suggested combining different test generation tools for improved performance. This study, along with the works discussed in Section 2.3, influenced our decision to explore Monkey-like frameworks for our research.

A recent study [11] highlights the main reasons to choose a test generation tool that could perfectly control the actions performed instead of performing pseudo-random events. The authors conducted a comprehensive examination of the limitations associated with Monkey, a widely used testing tool. Their study focused on analyzing Monkey's performance across a benchmark of 64 apps, aiming to identify common limitations that hinder the

tool from achieving optimal coverage results. Furthermore, the authors assessed the potential improvement in coverage that could be achieved by eliminating these identified limitations. That's why we didn't choose Monkey as our reference Framework, but a more evolved tool, always from the Android family, as it will be described in chapter 4.

Subsequently, we explore several automated testing tools, starting with *Mobicomonkey* [4]. This tool, building upon Monkeyrunner, serves as an extension of the Monkey Framework, offering enhanced capabilities. It enables developers to test applications against custom or auto-generated contextual scenarios, facilitating bug detection through the emulator. Notably, it reports the connection between bugs and contextual factors for later reproduction. By utilizing Android SDK and Logcat, *Mobicomonkey* injects events and captures traces, excelling at handling unforeseen scenarios.

Considering the Software Testing as a Service (TaaS) methodology [82], a cloud-based framework called Automated Mobile Testing as a Service (AM-TaaS) is introduced. It utilizes cloud infrastructure to provide on-demand testing services for mobile applications based on AQuA's test criteria [8]. An experiment emulating nine Android devices tests the "OTA Install" criterion under virtual machines and cloud infrastructure.

The automated tool *CRASHSCOPE* [48] specializes in identifying potential sources of crashes. Employing systematic input generation informed by static and dynamic analyses, it aims to trigger crashes. Upon detection, *CRASHSCOPE* generates a detailed crash report with screenshots, reproduction steps, exception stack trace, and a replayable script for automatic reproduction on a target device.

Addressing the challenge of test re-execution, *Paladin* [45] offers a solution by creating reproducible test cases for Android apps. Leveraging a GUI model based on the structure of the GUI view tree, it identifies equivalent app states, ensuring consistent behavior in different test executions.

Two tools facilitating test development without code are *MT4A* [15] and *Barista* [22]. *MT4A* is a framework allowing the creation and execution of tests for Android applications without programming skills, supporting automated tests based on predefined actions and data injection into device sensors. On the other hand, *Barista* enables testers to create platform-independent test scripts easily and run them on multiple devices and OS versions without modifying the app or runtime system.

Lastly, emerging techniques leveraging machine learning for automated testing are introduced, exemplified by *AppFlow* [28] and *Humanoid* [41]. These tools pioneer the integration of machine learning, with *AppFlow* synthesizing robust and reusable UI tests through

machine learning and *Humanoid* using deep learning to guide test input generation based on learned human interaction traces in black-box Android app testing.

2.3. Related Work

In the selected publications [46] and [47], the authors focus on network data collection, analysis, and Android Automation Testing. Their work addresses the management of network traffic generated by smart devices and companion apps to enhance privacy and security. The authors classify network traffic as critical or non-critical, with critical traffic being essential for smart devices to function properly. They introduce two complementary methods, *IoTrigger* for generating network traffic and *IoTrimmer* for blocking non-critical traffic. The Monkey framework provided by Android Studio is leveraged for generating network traffic from smart devices, serving as a foundational approach for the current thesis. The authors also propose a method for handling potential failures in the interaction between smart devices and companion apps, utilizing screenshot matching with ImageMagick [29]. While the same idea is avoided in the current work, a specific code is developed to address this issue.

3 | Project Overview

This section outlines the primary methodology employed in this work. We begin by explaining the conceptual framework, including the underlying idea, motivation, and rationale for the proposed framework. The initial segment provides a contextual overview, defining the environment under scrutiny. This includes a detailed exploration of the identified problem and the overarching objectives derived from it.
Subsequently, we present the devised solution crafted to realize our objectives and meet the specified requirements. This encompasses a comprehensive explanation of the developed framework.
To facilitate a thorough comprehension of our tool, we introduce pertinent theoretical concepts. Emphasis is placed on clarifying the key distinctions between the conventional framework and our specific contextual framework.
Lastly, we elucidate the project workflow, offering a comprehensive overview. Further details on the implementation and a breakdown of software/hardware aspects are expounded in Chapter 4.

3.1. Problem Description

In general, forensic investigation techniques are the application of science to criminal and civil laws. It is a broad field utilizing numerous practices such as the analysis of many potential sources of evidence such as DNA, fingerprints, bloodstain patterns, firearms, ballistics, toxicology, and fire debris. Forensic scientists collect, preserve, and analyze scientific evidence during an investigation. While some forensic scientists travel to the scene of the crime to collect the evidence themselves, others occupy a laboratory role, performing analysis on objects brought to them by other individuals. Others are involved in the analysis of financial, banking, or other numerical data for use in financial crime investigation and can be employed as consultants for private firms, academia, or government employees. We are strictly related to the laboratory role of the investigation.

Our work is located in the IoT Forensics domain, a branch of Digital Forensics that deals with IoT devices. More specifically, in the network forensics scheme, we have to collect

evidence by inspecting network traffic exchanged between a smart device and its mobile app. In our scenario, the evidence is a dataset containing network traffic records. One of the main problems with IoT network traffic analysis is the lack of labeled datasets, which contain both the network traffic and the device activity. We conventionally named the network packets associated with the device activity as *EVENTs* (e.g. a bulb turning on, a camera rotating its view, etc), while all the rest of the traffic is *spontaneous*. With the term *spontaneous* we mean network packets produced when a smart device and its companion app do not actively communicate.

Manually building such datasets requires a lot of time and human effort, requiring the user to interact continuously with the devices and take note of the correct times in which the activity is performed. The full procedure can be automated and this is the main purpose of our work. The devices will be connected to an access point equipped with a packet sniffer program (e.g. Wireshark, Tcpdump, etc...) able to capture a network traffic dataset. This dataset along with the activity log is used to perform the first kind of data analysis, and it will be used as input for machine learning methods or other forensics processing techniques in future research. We will give a basic grasp of potential data analysis that could be done with the dataset collected. This is the secondary purpose of the thesis. The analysis has been focused on characterizing the network traffic, trying to answer questions like *"Does it exist a kind of spontaneous traffic generated by the smart device when EVENTs do not trigger it?"*, *"Is there a pattern that can be used to discern user EVENTs from spontaneous traffic?"*. In the following sections, we will describe our solution for the automation task and some theory behind it.

3.2. Proposed Solution

To answer the previous questions, we need a methodology able to automate the gestures made by a real person on the screen of her/his smartphone. This method has to interact directly with the UI items displayed in the foreground of the smartphone's screen. The term ITEM is referred to a mobile UI element available to developers. They could use those elements to develop the GUI of the mobile app. This is the entry point for the service or business they want to supply. This method has to inspect those items, navigate through them, identify the targets, and finally interact with them, simulating a gesture that a real person typically does. This is because by manipulating the user interface of the companion app, the final user triggers a change of state of the smart device.

This process is known in the literature as Mobile App Testing. The essence of this process is to use a particular component of the mobile app to simulate the gesture of a human

being over the screen of his/her mobile device. This framework is the core element around which this master thesis is developed. We have brought to bear it for the automation part of the mobile device gestures. In this way, the smart device, controlled by the automated mobile app, has been induced to different states, for instance, switching on/off a smart plug or increasing/decreasing the saturation of a smart bulb. We developed AutomIoT, a framework able to automate the EVENTs generation and write an activity log to keep track of them. Then, the network traces generated by those different states have been collected, cleaned, and analyzed.

Now, we will dig into the details of the Mobile App Testing framework, providing some theoretical aspects before moving into the structure of our project and how we conceived it.

3.3. Mobile App Testing

Mobile application testing is a software testing process for applications designed for handheld devices. Typically, even before the mobile app is released into the app marketplace (e.g., App Store, Google Play), these apps are tested to meet all functional and non-functional requirements, business requirements, and user expectations. The main aim of mobile app testing is to improve the quality of mobile apps and deliver a great UX. It is an integral part of the development process. [79]
Before giving a more in-depth overview of the testing part, we start from the basis by introducing some concepts about the app, the device typology on which they run, and finally, an overview of possible gestures that can be used to interact with the screen of the device.

3.3.1. Mobile Application

A mobile application, most commonly called an app, is software designed to run on a mobile device, such as a smartphone or a tablet. Mobile applications frequently provide users with similar services to those accessed on computers. There are three types of apps in the mobile universe today: [75]

(i) **Native:** They are designed to run specifically on a mobile operating system, having access to internal features of the device (e.g., camera, GPS, and accelerometer). (e.g. Whatsapp, Spotify)

(ii) **Web-Apps:** here part or all of the application is downloaded from the web when the application is run. This type of application can be accessed by virtually any

device that has internet access through a web browser (e.g., Google Chrome or Firefox). In contrast, it does not have direct access to internal device features such as those cited in the native application category (e.g., Google Docs, Netflix).

(iii) **Hybrids:** contain characteristics of the two categories mentioned above. They must be installed, have access to internal features of the device (such as native applications), may have some of their content downloaded from the web, and may be based on the HTML5 language (as well as web applications). Mobile application development differs considerably from traditional software development (Gmail, Twitter/X).

Devices Overview

Unlike static analysis of mobile apps, testing requires running apps on an execution environment such as a real device or a virtual device.[76].

What is a real device? Real testing devices are the various models of mobile handsets used to run the website or app to test its functioning and behavioral patterns. These are actual handsets that the end user would use.

What is a virtual device? Virtual devices are software program that provides simulation for most of the key features of an actual mobile device. These are of two types:

- ***Emulators:*** Emulator is software that mimics both the hardware and software of the target device on the computer.
- ***Simulators:*** Simulator is a software that only mimics software and not hardware and runs certain programs built for a different OS.

Which one do we choose? The answer lies in what you want to test. If you are looking to test the performance of your app, you must go with real devices. Real devices are the actual mobile devices that the end user would be using to run the application. They provide accurate results and allow testing in the same condition as the end user. Emulators are preferred when testing the mobile's external behavior, such as calculations, making transactions, etc. Simulators, on the other hand, are less reliable than emulators and are not suitable for debugging. Also, the benefits of emulators/simulators compared to a real device could be recapped as shown in figure 3.1.

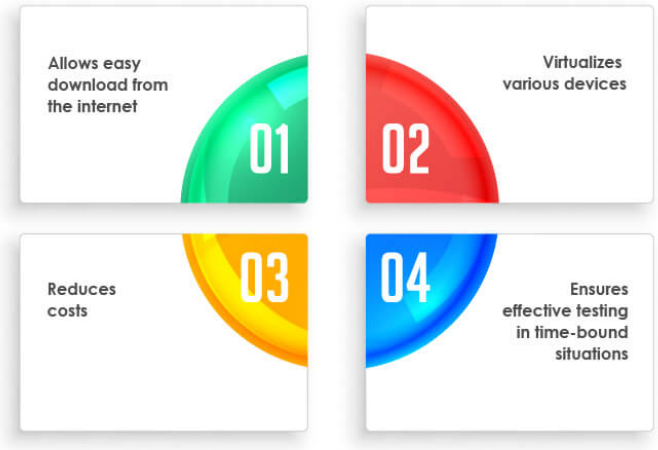

Figure 3.1: Benefits of leveraging simulators and emulators

As we will see in chapter 4, we prefer and choose to run our tests inside an emulator since it does not require additional hardware and can be replicated easily, and also because we didn't have to test the performance of the mobile app, but we only needed a tool to automate the changing of states of the smart device and gathering network data. Benchmarking the two could be a case study for a future thesis. For this purpose, we use a laptop to emulate Android devices.

Gestures Overview

Gestures are the first gate to inject input commands on our mobile devices. They are the first way in which we interact with them. There exists a lot of those actions, mainly divided by single or multiple gestures. The classification goes even further, but for the sake of our research, we will only scratch the surface. In the picture depicted below there is a collection of the main gestures that can be carried out over the screen of a mobile device.

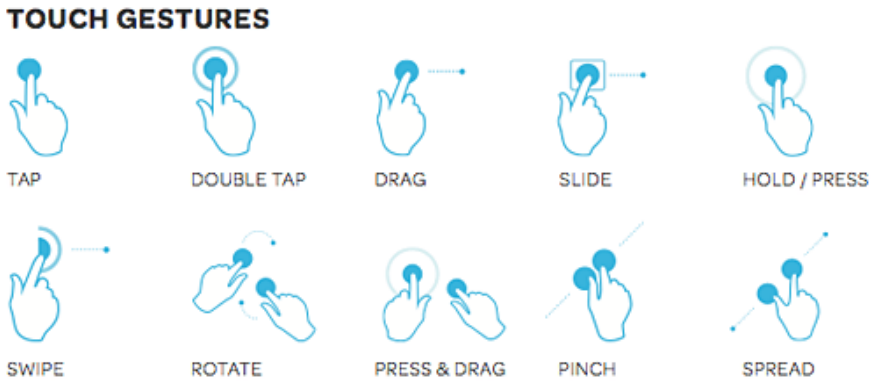

Figure 3.2: Example of Gesture icons

We will enter into the details concerning the ones that we will treat later in the chapter 4.

- ***TAP:*** Also known as touch, or click. Touch the screen surface with a fingertip.
- ***DRAG:*** Move fingertip over surface without losing contact. This action could be performed in any direction indistinctly. It is a fine gesture, slower, more controlled, and typically has an on-screen target to reach with an element involved in this action.
- ***SLIDE:*** It refers to a gesture where the user moves his/her finger from left to right or right to left on the screen.
- ***SWIPE:*** Similar to a slide, a swipe is a gesture where the user moves their finger across the screen. However, a swipe typically involves a more rapid and forceful movement in a specific direction, such as up, down, left, or right. Swiping is often used to scroll through content. It is a gross gesture, faster, and typically has no on-screen target.
- ***SCROLL:*** it is a vertical or horizontal swipe in a single direction within the screen surface. Concerning a swipe, the scroll is a slower one.

Many more gestures will not be considered here, since they are out of the scope of this work, so they haven't been taken into account. For this reason, they are omitted from the previous list.

3.3.2. Classification

Mobile Application testing has a very deep and well-structured classification. In the following, we will glimpse the main categories and explain what we selected for our work.

Types

In general, there are three types of testing: White-box testing, Black-box testing, and Grey-box testing.

White-box testing is a scenario in which the software is examined based on the knowledge of its implementation details. It is usually applied by software developers in the early development stages when performing unit testing. Another common usage scenario is to perform thorough tests once all software components are assembled (known as regression testing). This usually means that we are actively developing the mobile app and we have all the knowledge that we need since we are the owner and maintainer of the code.

Black-box testing , on the other hand, is a scenario where internal design/implementation of the tested object is not required. Testing is based on an analysis of the specification of the component or system. If a Mobile app testing process only requires the installation of the targeted app, we reasonably put it under this category. The application is tested without looking at the application code and logic. The tester has specific test data to input and the corresponding output that the application should produce, and inputs the test data, looking for the program to output data consistent with what the tester was expecting. This method of testing can be applied virtually to every level of software testing: unit, integration, system, and acceptance.

Grey-box testing is a trade-off between white-box and black-box testing. It does not require the testers to have full knowledge of the source code as white-box testing needs. Instead, it only needs the testers to know some limited specifications, like how the system components interact. In our work, we will conduct black-box testing, and we will choose frameworks that allow the use of this type. This is because we don't have access to the mobile app source code, which is indeed proprietary software.

Subjects

There are different types of tests depending on the subject. [79]

- **Functional testing:** does my app do what it should? It's performed to evaluate

if a component or system satisfies functional requirements. [23]

- **Non-Functional testing:** It evaluates that a component or system complies with non-functional requirements. [24]
- **Usability testing:** It is used to evaluate the degree to which the system can be used by specified users with effectiveness, efficiency, and satisfaction in a specified context of use. [27]
- **Performance testing:** Does it do it quickly and efficiently? It's used to determine the performance efficiency of a component or system. [25]
- **Compatibility testing:** does it work well on every device and API level?
- **Security testing:** It's used to determine the security of the software product. [26]

We performed functional tests since we were verifying the correct execution of the UI elements in charge of changing the state of the smart device.

Approaches

In particular, the following testing approaches are used to run tests in a mobile development project: [75]

- **Emulation-based testing:** involves using a mobile device emulator (e.g., device simulator), which creates a virtual machine version of a mobile device for study on a personal computer.
- **Device-based testing:** this approach requires the creation of a test lab and the purchase of physical mobile devices, which consequently makes the process more expensive than the previous approach, but can verify device-based functions, behaviors, and QoS parameters that other approaches cannot.
- **Cloud testing:** This approach consists of a cloud of mobile devices that can support testing services on a large scale. Hence, several mobile device users can configure their required test environments through a rental service model.
- **Crowd-based testing:** offers benefits without investing resources to build a test lab or buy mobile devices.
  However, there is always the risk of reaching a low quality of tests, which may even have uncertain deadlines, which in the industry proves to be a terrible feature, given the short periods for delivering results and the pressure to meet deadlines.

In this case, since we have chosen an emulator, we have used an emulation-based testing

approach.

Scopes

Tests also vary depending on size or degree of isolation:

- Unit tests or small tests only verify a very small portion of the app, such as a method or class.
- End-to-end tests or big tests verify larger parts of the app at the same time, such as a whole screen or user flow.
- Medium tests are in between and check the integration between two or more units.

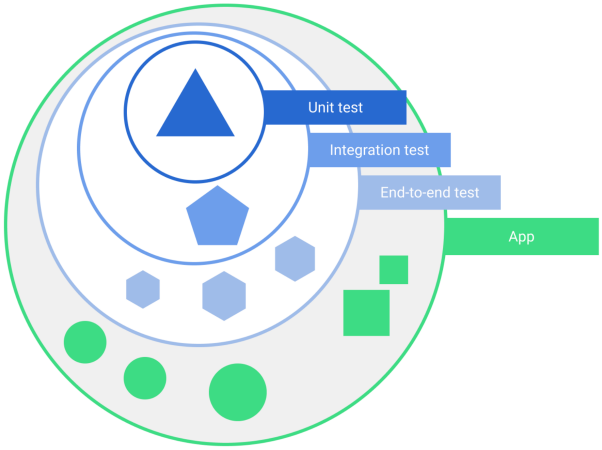

Figure 3.3: Test scopes in a typical application.

Unit Testing

In this work, it's important to describe and distinguish between unit local tests and unit instrumented ones, since we will use the latter. Let's see a more in-depth description and finally let's make a comparison between them.

Local (Unit) Test Unit Tests will ensure that the small units(a set of 1..n classes/functions) of an application work as expected in isolation validating the inputs and outputs. Unit testing is usually applied at the beginning of the development of Mobile apps, which are usually written by developers and can be taken as a type of white-box testing. Unit testing intends to ensure that every functionality, which could be represented as a function or a component, works properly (i.e., by the test cases). The main goal of unit testing is to verify that the implementation works as intended. Often Unit tests are referred to as "local tests" or "local unit tests". The main reason for this seems to be that a developer

wants to be able to run tests without a device or an emulator attached. Unit tests cannot test the UI for your app without mocking objects such as an Activity

Instrumented (Unit) Testing To instrumentate something in the context of software development is about the usage/creation of tools that provide ways to manage, measure, or control a system. Instrumentation tests are used for black box testing. It is used to test the GUI of the application along with its functionality in a real environment. Instrumentation tests run on a device or an emulator. In the background, the app is built and installed alongside a test app that injects commands and reads the state. This test app will control your app, launching it and running UI tests as needed. Instrumentation tests can be used to test non-UI logic as well. They are especially useful when you need to test code that has a dependency on a context.

Unit versus Instrumented Instrumented tests are usually UI tests, launching an app and then interacting with it. Unit tests, also called local tests, execute on your development machine or a server, so they're also called host-side tests. They're usually small and fast, isolating the subject under test from the rest of the app. [57]

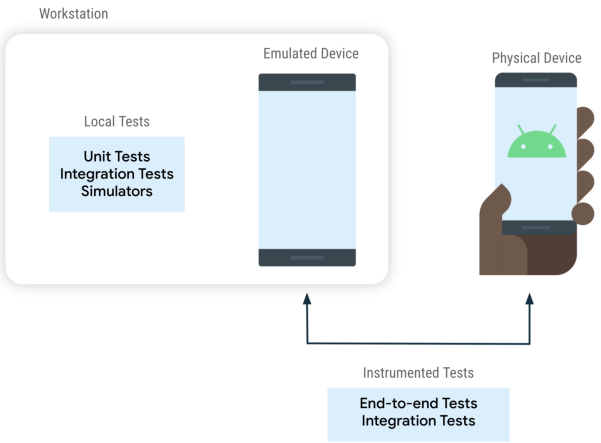

Figure 3.4: Different types of instrumented tests.

Unit tests are the test cases written for testing code written by developers and Instrumentation tests are the test cases written for testing the UI and functionality of the application. In short, Unit tests are associated with white box testing and instrumentation tests are associated with black box testing. We will use Instrumented Tests for this last reason.

Finally, we can view how it is possible to automate a test of this kind.

3.3.3. Mobile Automation Testing

Testing user interactions is a crucial step that can help the developer avoid future users having unexpected results or a poor experience when interacting with your app. One approach to UI testing is to simply have a human tester perform a set of user operations on the target app and verify that it is behaving correctly. However, this manual approach can be time-consuming and error-prone. A more efficient approach is to write your UI tests such that user actions are performed in an automated way. The automated approach allows you to run your tests quickly and reliably in a repeatable manner. UI tests launch an app (or part of it), then simulate user interactions, and finally check that the app reacted appropriately. They are integration tests that can range from verifying the behavior of a small component to a large navigation test that traverses a whole user flow. They are useful to check for regressions and to verify compatibility with different API levels and physical devices. UI test is the term used for any test that verifies the correct behavior of a UI.

3.3.4. Framework Comparison

The state-of-the-art presents many different tests able to accomplish this process, mainly used to perform in-depth mobile app performance evaluation to have a solid application to deploy in production. These tests are meant not only to verify the correct behavior of the developed mobile app but also to understand how the mobile app reacts to unexpected user inputs to increase the security of the system. Indeed, we previously defined our tests as only functional.

This work is meant to validate and use only a subset of all the possible actions that can be triggered by the mobile app testing techniques. We only need the actions capable of triggering a change of state in the smart device. They are a finite subset strictly related to the gestures involved, and described before. This subset demands precisely the actions that permit and trigger the interaction between the mobile app and the controlled smart device (e.g. change the color of a smart bulb, turn on a plug, etc...) so that it will generate network traffic to sniff afterward.

This is the reason why it is not possible to fully automate all the processes. By all the process we mean the research of the gestures and their execution. In literature, there exist some advanced tools able to automate the former element, but in this work, we only need to automate the latter as described in the testing automation section. 3.3.3.

To recap, in our work, it is needed a manual selection of the subset of actions that trigger

EVENTs exploiting the corresponding gestures, creating the code that exploits them, running it, and finally automatically generating EVENTs to gather. Here the automation part involves the task execution, not the gesture recognition and selection.

3.3.5. Mobile App Testing Track

What are the steps involved in mobile application testing? [79]

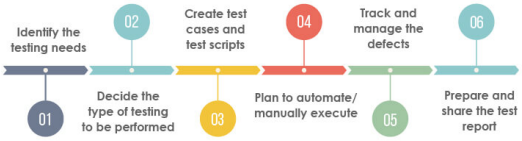

Figure 3.5: Mobile App Testing Steps

Identify the testing needs

The first step is to identify the testing needs. Testers need to decide what is to be tested in an app and decide the scope and the test coverage.

Decide the type of testing to be performed

Once all the testing needs are identified, testers need to identify the type of tests to be performed on the mobile apps. Various types of tests are performed on new mobile apps, including functional testing, regression testing, performance, and load testing, interruption testing, localization testing, speed testing, usability testing, security testing, accessibility testing, etc.

Create test cases and test scripts

The next step involves creating test cases and automated test scripts. Testers should ensure that the test cases should cover all the critical areas and functionalities of an app.

Plan to automate/manually execute

Based on the testing needs, testers need to decide whether to automate or manually execute the test. Testers need to leverage automated mobile testing tools as per testing requirements to perform mobile automation testing to quicken the releases and ensure faster time-to-market.

Track and manage the defects

Once the testers execute all the test cases, they need to identify the issues in an app and fix them as soon as they are identified. Lastly, testers must prepare a test execution report detailing all the passed/failed test cases, bugs identified, bugs resolved, and other remarks. The report must be shared with all stakeholders to help them make informed decisions. However, in today's fast-paced world, for an app to be successful in the market, it should work seamlessly on all hand-held devices available in the market. But, testing mobile apps on all possible devices is difficult for testers. Therefore, testers need to leverage a combination of real devices and virtual devices (Simulators and Emulators) for mobile app testing. In this section, we have seen the main steps that regard the development of a mobile app test. For our work, we will take into account up to the planning part. The reason related to this choice is motivated by the fact that our goal is to create a dataset exploiting this framework, not verify bugs of the mobile application, that we are inspecting. We only need to collect the dataset, after this our job is done, and we can skip to the analysis part.

3.4. Project workflow

In this section, we have highlighted the main types of devices that we needed for this thesis, and we provided a concrete implementation of the steps defined in the section 3.3.5, related to our project.

3.4.1. Devices

The devices needed to achieve the goal of this thesis are the following:

- One or more smart devices.
- A computing device, is needed to find the UI element to map, develop the UI test, and load the UI test binary to the mobile device.
- A mobile device running the smart device mobile management app.
- An access point, that acts also as a network packet analyzer and sniffer.
- A network dump traces post data processor.

Those are the main actors that are needed and will be described in the next chapter (chapter 4) more in detail.

3.4.2. Workflow

Let's put into practice the definitions of the section 3.3.5.

Identify the testing needs

We need to test the functional behavior of the gestures able to trigger a change of state in the controlled smart device (Test Coverage). As for scope, we selected the Unit Testing one, in particular the Instrumented Test.

Decide the type of testing to be performed

We will perform Functional tests with a black-box testing framework having as a target 2 mobile apps related to 4 smart devices.

Create test cases and test scripts

We have developed a series of tests with increasing complexity. We have started by inspecting the mobile app through a specific tool with 2 objectives, the first was to discover all the possible ways in which the state of a smart device could be changed (e.g. on/off, different color hue, different color saturation, different device orientation, etc...). The second was to collect all the mobile UI elements able to induce those changes.

Now that we have all the ingredients for our recipe, we can make our cake, so we developed the first test by selecting one smart device, its management app, and one change of state to apply. We ran it and verified the correctness of the tests. We performed the same step for all the possible changes related to the same device and app, and then we did the same thing for all other devices and mobile apps involved. Every test was responsible for verifying a single specific component, so we ran a test for every single change of state for all devices.

Plan to automate/manually execute

Finally, we created AutomIoT, our automation tool. The first step was to create a test able to map all the possible gestures that we needed for all mobile and smart devices. Then, we ran the test in a deterministic way defining an arbitrary order of execution, and subsequently, we introduced a random ordering in our test. We started to run tests for a certain number of iterations. We will see that by choosing the correct number of iterations, we will perform the test for a certain amount of time.

Data Gathering

After we tried the test a couple of times to verify the correctness of our tool, we launched our "alpha" test. During the execution of the test, we enabled the network packet sniffing feature of our Access Point (AP). Another important element is that while the test was running we collected a ground truth file to verify the gesture performed, the timestamp of execution, the mobile app, and the smart device involved.

Data Analysis

In this final part, we compare the ground truth file records with the network traces records to associate the EVENTs collected with the network part to answer our initial questions.

4 | Implementation Details

This part of the work will describe the architecture that we built, its main components (hardware and software used), how they work together, and finally the implementation of AutomIoT, the data gathering, and data analysis.

4.1. Architecture Overview

This section describes the architecture of our system.

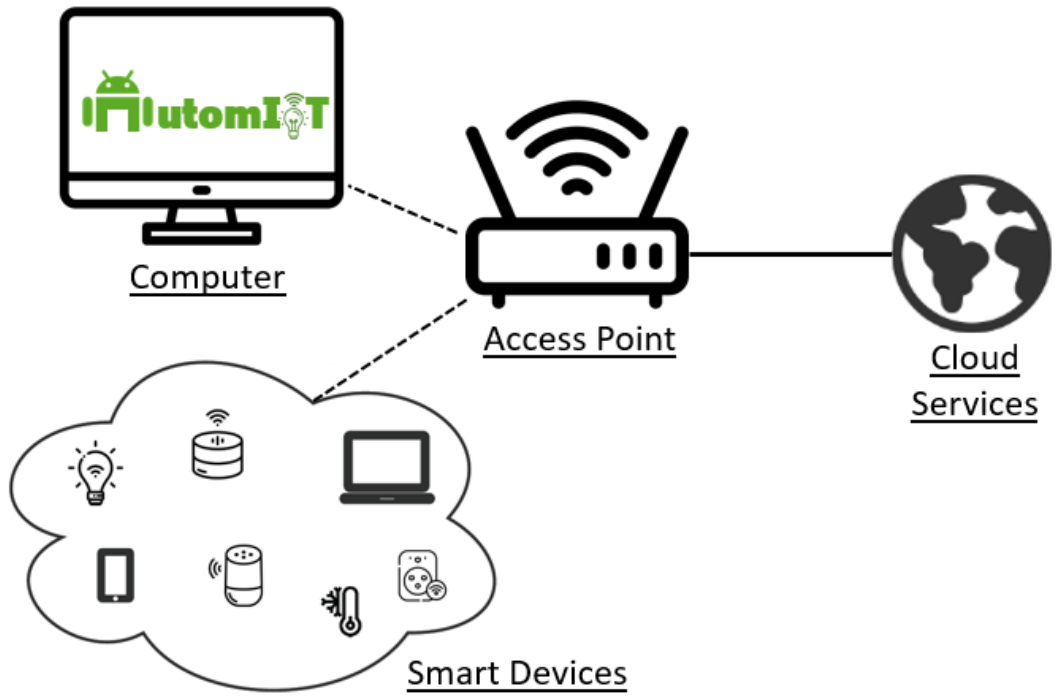

Figure 4.1: Overall Architecture

This diagram points out the main hardware devices adopted, and how they communicate together. The Wi-Fi links are depicted with black dashed lines, while the wired ones are shown with black solid lines. The former links concern only the traffic among smart devices, the computer, and the AP, while the latter ones refer to the requests that go from

the AP to the cloud services.
The architecture is constituted by 2 building blocks:

- ***Cloud Server:*** The various brands have exploited some cloud computing services for the development of the backend of their infrastructure. With the term backend, we mean the server part of a client-server architecture. The mobile apps are the client part of the architecture. Here we model those cloud part as a black-box since the source code and functioning is proprietary and not available. We could only infer that the cloud services manage HTTPS requests sent by smart devices and mobile apps. In particular, it manages the communication between a smart device and its mobile app (and vice versa). There isn't direct communication between them, but it is brokered by a cloud service that acts as a server. This information was derived by analyzing the network traffic collected. Indeed with Tshark, we could see some FQDNs related to the cloud services for the backend part, and also FQDNs related to other services, like Facebook, Bing, Google, etc..., or other third-parties not strictly related to the app's first objective, out of the scope of this work. The cloud part also stores and controls the current state of the smart devices registered using the mobile app.
- ***ANTLab setup:*** All the smart devices, and the laptop are connected to the AP through a WLAN. All the requests incoming to the AP are HTTPS requests apart from the HTTP requests that come in from the laptop, related to the Web App console management of the AP. The AP manages the network traffic towards cloud services.

As it will be described later in detail, the Computer played more than one role concerning the devices defined in the section 3.4.1 of the previous chapter. Indeed it has been used as a computing device for UI Test automation, as a mobile device emulator, and finally as a post-data processor for the data gathering and data analysis part.

4.2. Automation Overview

This section will explain the details of the AutomIoT framework implementation, the platform being used, the smart device setup, how emulation was performed, the software that was used, how we created and automated the UI Instrumented Test, and how we finally collected and analyzed the network traffic data.
We created a step-by-step procedure to reproduce our setting, using the AutomIoT framework to perform identical or similar labels network dataset collection. [71]
The source code of AutomIoT could be examined on GitHub. [70]

4.2.1. Platform

There are a lot of tools that could simulate the gestures of a human being over the screen of a smartphone. Mainly, in the state of the art we have seen tools related to the Android world. In particular, we utilized as a starting point the papers of Mandalari [46], [47] that exploit the Monkey framework to simulate interactions with the mobile device. This is not the only testing tool provided to Android Developers. We will go into the details of each one to justify our aims in section 4.2.4.

As a result of wide adoption in research, and wide adoption in the market concerning competitors (e.g. IOS), we have chosen the Android Platform and Android Studio as a focal point for development ends. Android Studio [54] is the official IDE provided to developers, and it has been the heart of our UI Test Automation.

4.2.2. Smart Device Setup

The setup of a smart device comprehends 2 phases.

PHASE 1: It is needed to plug in the smart device to the power source.

PHASE 2: The last thing to do is associate the smart device with the mobile app. To perform this operation it was used a real mobile device. We needed a real device since it was not possible to perform the association directly with the emulated device. It could be an issue related to the network settings of the emulator that we utilized.

The association procedure is as follows:
After downloading the companion app, depending on the smart device that we would like to pair, and logging in, it has been needed to switch on/off the smart device 3 consecutive times to make it blink intermittently (this operation was accomplished with the hard switch button), then we have to connect the real device to the Wi-Fi connection hosted by the smart device, add the new device to the mobile app, and finally associating and connecting the device to the AP WLAN. Now the smart device is ready to be used by the emulator and it is connected to the correct WLAN to gather network data later on.

4.2.3. Mobile Device Emulation

Android Studio allows to emulate a real Android device by using Android Emulator [53].

What is Android Emulator?

It allows you to run emulations of Android devices on Windows, macOS, or Linux machines. The Android Emulator runs the Android operating system in a virtual machine called an Android Virtual Device (AVD) [55]. The AVD contains the full Android software stack, and it runs as if it were on a physical device. Figure 4.2 is a diagram of the Android Emulator's high-level architecture.

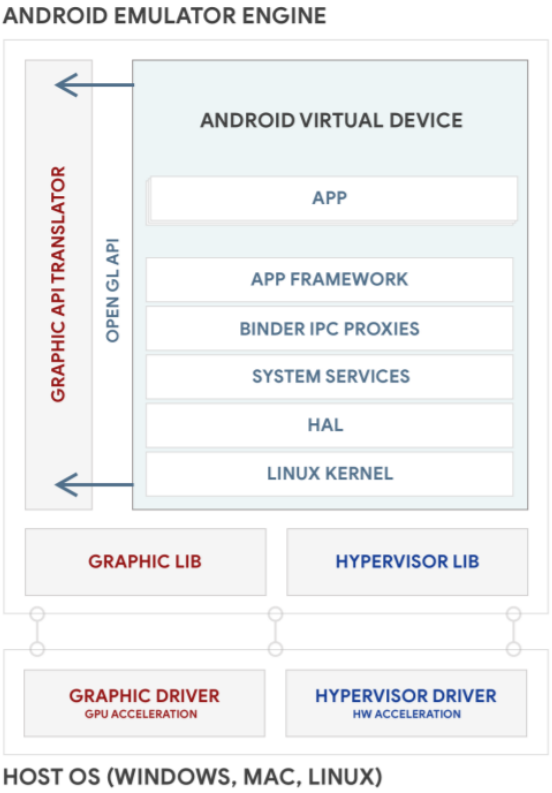

Figure 4.2: Emulator Architecture

The Android Emulator uses the Quick Emulator (QEMU) hypervisor [17]. The different emulators created in this way could be managed by 2 tools offered by Android Studio, which are Virtual Device Manager (GUI) and AVD Manager (command-line).

What is Quick Emulator?

When used as a machine emulator, QEMU can run Operating Systems and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your PC). By using dynamic translation, it achieves very good performance. When used as a virtualizer, QEMU achieves near-native performance by executing the guest code directly on the host

CPU.

What is Android Virtual Device (AVD)?

An AVD contains a hardware profile, system image, storage area, skin, and other properties.

Hardware profile defines the characteristics of a device as shipped from the factory. The Device Manager comes preloaded with certain hardware profiles, such as Pixel devices, and you can define or customize the hardware profiles as needed.

System images: A System Image labeled with Google APIs includes access to Google Play services. A system image labeled with the Google Play logo in the Play Store column includes the Google Play Store app and access to Google Play services, including a Google Play tab in the Extended controls dialog that provides a convenient button for updating Google Play services on the device. To ensure app security and a consistent experience with physical devices, system images with the Google Play Store included are signed with a release key, which means that you cannot get elevated privileges (root) with these images. If you require elevated privileges (root) to aid with your app troubleshooting, you can use the Android Open Source Project (AOSP) system images that do not include Google apps or services.

Storage area: The AVD has a dedicated storage area on your development machine. It stores the device user data, such as installed apps and settings, as well as an emulated SD card. If needed, you can use the Device Manager to wipe user data, so the device has the same data as if it were new.

Skin: An emulator skin specifies the appearance of a device. The Device Manager provides some predefined skins. You can also define your own, or use skins provided by third parties.

Setup

With the help of Android Virtual Device Manager, we picked the following specifications for our virtual Device. The following choices were arbitrary. It is suggested to use the same one.

NAME	Automator
DEVICE MODEL	Pixel 2 (1080x1920 xxhdpi)
SYSTEM IMAGE	Google APIs Intel x86 Atom_64 System Image with root privileges on the file system e.g. S,31,x86_64,Android 12.0(Google API)
OS:	We used Android OS version 12.0 (S) x86_64 with Android SDK 31 that was the last stable one released at the time.
HW PROFILE:	2 vCPU, 4096 MB RAM, VM heap 1024 MB, internal storage 2048 MB as managed SD Card.
ADVANCED SETTINGS:	In the Camera Section set Front to None and Back to None. In the Emulated Performance section select Cold Boot as the Boot Option. In Device Frame, check the checkbox Enable Device Frame and select no skin.

Finally, we have to install the mobile apps inside the emulator, in section 4.2.6, we will see 2 of them used to test the AutomIoT framework.
We want to highlight 2 possible choices here. It is possible to install the mobile apps by using the Google Play Store provided by the virtual device creating a Google account if necessary or using an existing one, or it could download the corresponding APK (e.g. Tapo.apk, EZVIZ.apk) file to be installed into the emulated device exploiting ADB Android Studio CLI [52] from websites as APKMirror [6] or APKPure [7]. We have decided to install directly the APK files downloaded from APKPure.

4.2.4. Instrumented Test

The main actor in developing something Android-related is Android Studio, which is the official IDE provided to developers. Android Studio has 3 frameworks to accomplish the task of creating an Instrumented test:

Monkey [58], is an old tool to generate mainly pseudo-random events to see the robustness of the mobile app tested, this is used for multi-application black-box testing. It is command-line based, indeed it exploits the ADB tool to launch the test.

There is MonkeyRunner [59] (deprecated and unmaintained in favor of UI Automator),

which using a Python API allows us to write a black-box testing and launch a Monkey test through a command-line tool inside the mobile device. Compared to Monkey which performs random events, the tests could be piloted accordingly.

Then, we have a library called Android Jetpack.

Android Jetpack Jetpack is a suite of libraries to help developers follow best practices, reduce boilerplate code, and write code that works consistently across Android versions and devices so that developers can focus on the code they care about. Jetpack includes various frameworks that provide APIs for writing UI Tests:

- Espresso [56] testing framework provides APIs for writing UI Tests to simulate user interactions with Views within a single target app. A key benefit of using Espresso is that it provides automatic synchronization of test actions with the UI of the app you are testing Espresso detects when the main thread is idle, so it can run your test commands at the appropriate time, improving the reliability of your tests. Espresso is a bit more lightweight compared to UI Automator. Slightly more grey-box testing compared to UI Automator since it could be accessed through the internal view app resources. This is recommended for "Testing UI for a Single App". If you are testing only a single application, that you are developing, then Espresso is the best choice.
- Jetpack Compose provides a set of testing APIs to launch and interact with Compose screens and components. Interactions with Compose elements are synchronized with tests and have complete control over time, animations, and recompositions.
- UI Automator [60] is a UI testing framework suitable for cross-app functional UI testing across system and installed apps. The UI Automator APIs allow you to perform operations such as opening the Settings menu or the app launcher on a test device, and let you interact with visible elements on a device, regardless of which Activity is in focus. [60]. The UI Automator testing framework is an instrumentation-based API and works with the AndroidJUnitRunner test runner. It's well-suited for writing opaque box-style automated tests, where the test code does not rely on internal implementation details of the target app. UI Automator is powerful and has good external OS system integration e.g. can turn WiFi on and off and access other settings during tests, but lacks detailed view access so one could say it may be more of a pure black-box test. If you are testing more than one application or its integration with other applications or systems, then the best choice is UI Automator, regardless it's a self-developed application or not.
- Robolectric lets you create local tests that run on your workstation or continuous

integration environment in a regular JVM, instead of on an emulator or device. It can use Espresso or Compose testing APIs to interact with UI components.

UI Automator was our choice because it allows us to write black-box testing and also to test more apps at the same time, and this is a requirement for us since we have to manage 2 distinct mobile apps. Moreover, we take inspiration from the papers of Mandalari [46], [47], and ruling out Monkey since it generates random actions and since we need total control over the actions that will be performed, and ruling out MonkeyRunner since is unmaintained.

UI Automator: How it works?

To automate an Instrumented test, it is needed to identify, detect, and select the UI Elements part of an app GUI. Those components can induce several effects. As pointed out in the 3.2, in this work only elements that will trigger a change of state in the final smart device will be taken into consideration, which we named EVENTs; Examples of UI Elements that produce this change of state are buttons, checkboxes, sliders, switches, radio buttons, images, and many more.

The first step is to detect them manually trying to interact with the mobile application and verifying that the smart device has changed its state. For instance, a smart bulb switches from on to off. After this, exploiting a GUI inspection tool (e.g. Appium inspector for Android/iOS, UIAutomatorViewer for Android), it can be plugged into the screen of the device used, and it can be explored by searching for the UI Elements that are needed. With the term "to be plugged" it is meant that the GUI Inspection tool will be connected to the device through a socket and will be able to read the underlying XML from which the GUI is rendered. Reading the XML file, it is displayed a copy of the mobile device screen and a tree reflecting the XML contents. In this tree are contained TAGs, and they are the ones necessary to perform the UI TEST. The TAGs are keywords specified by the developer during the app development process, and can clearly identify UI Elements. Essentially, they are strings, such as resource IDs (identifiers), class names, pixel positions, text labels, etc...

For the pixel positions, UIAutomatorViewer provides a pair (e.g [x1,y1] [x2,y2]), with the former as the position of the most top left pixel UI Element inspected, and the latter as the position of most bottom right pixel UI Element inspected. In this case, an EVENT would be triggered by computing the center pixel given the aforementioned pair and using this pixel coordinate to perform a gesture over the mobile device screen. In Figure 4.3) we could see an example.

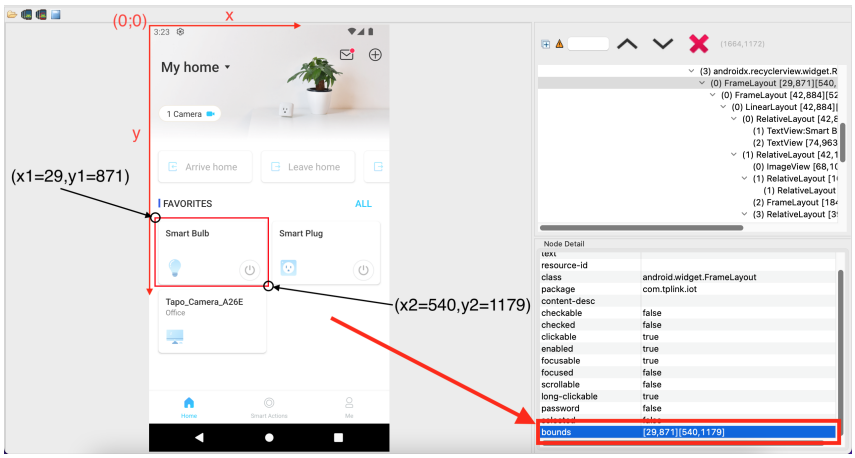

Figure 4.3: Bounds [x1,y1],[x2,y2]

Those TAGS then can be used as selectors in the UI Test to cause the change of state in the smart device. Let's see in the next paragraph the GUI inspection tool provided by Android Studio.

What is UIAutomatorViewer? It is a tool that provides a convenient visual interface to inspect the layout hierarchy (mainly Views) and shows the properties of UI components that are visible in the foreground of the device. This information lets you create more fine-grained tests using UI Automator. For example, we can create a UI selector that matches a specific visible property. [60]

What is an Android View? This class represents the basic building block for user interface components. A View occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for widgets, which are used to create interactive UI components (buttons, text fields, etc.). The ViewGroup subclass is the base class for layouts, which are invisible containers that hold other Views (or other ViewGroups) and define their layout properties. Figure 4.3 depicts an example of a UIAutomatorViewer GUI. It's like reading an HTML page on your browser and inspecting the code written, but in this case is an XML. The main properties used as selectors are the following:

- package
- class
- resource-id (unique)
- text

- bounds [x1,y1][x2,y2]

All those selectors would be used in the next step, to write the code needed to access the app automatically while running the test. Throughout this work, the selectors would be related to TAGs.

UI Automator: Test Development

The collection of TAG selectors has been employed to develop our test. We followed a series of steps to develop our test. The first one is using the emulator or a real device to verify the correct association between an action performed (e.g. a button click) and a changing of state (e.g. bulb switches on). This is done manually. In practice, we have conducted a series of functional tests to verify the functioning of the buttons.

After this "Pilot Test", we started to write our first piece of code in Android Studio. We have done the same kind of test, but automatically, launched the test from Android Studio. The test involves a specific action that triggers defined events. For instance, a click of a UI Element switches on a bulb or changes its color hue.

Before presenting the main components of our code, we have to introduce some references concerning the basic blocks to understand the meaning. Inside our Kotlin class, we have to specify at the beginning of the class the clause `@RunWith(AndroidJUnit4::class)` to perform an Instrumented test. The first block is characterized by the UiDevice object. UiDevice provides access to state information about the device. You can also use this class to simulate user actions on the device, such as pressing the d-pad or pressing the Home and Menu buttons. It is the controller for every single action that we can perform over the screen of the device. This object is the core of every action executed. The following illustrates an example of UiDevice instantiation.

```
val device: UiDevice =
↪  UiDevice.getInstance(InstrumentationRegistry.getInstrumentation()
```

After having defined a device, we could exploit the selectors found before. Through the method `device.findObject(...)` is possible to select the specific UI element that we need to stimulate, to trigger an EVENT. There is another way, that we used in the test to trigger an EVENT and it is through the method device.click(), but in this case, the selectors are not specific TAGs, but the pixel position of the UI element over the screen. Other than a click, is it possible to execute actions like drag, swipe, scroll, clickAndWait, etc... The following are possible usages of the `device.findObject(...)` method.

```
device.findObject(By.res(SmartObjResourceId.TAPO_SMARTBULB_STATE_BTN.rid)).click()
```

```
device.findObject(UiSelector().resourceId(RESOURCE_ID)).click()

device.findObject(UiSelector().resourceId(RESOURCE_ID)).swipeDown(INT_VALUE)

device.findObject(UiSelector().resourceId(RESOURCE_ID)).swipeUp(INT_VALUE)

device.findObject(UiSelector().resourceId(RESOURCE_ID)).dragTo(DEST_X,
↪  DEST_Y, N_STEPS)

device.click(randomPair.first,randomPair.second)

device.findObject(By.desc(obj.app.appName)).clickAndWait(Until.newWindow(),
↪  SmartObjDelay.DELAY_WINDOW.delay)

device.findObject(By.res(SmartObjResourceId.TAPO_SMARTBULB_STATE_BTN.rid)).click()
```

We used Kotlin as a programming language to develop our test. It is a modern language designed to interoperate fully with Java, and the JVM version of Kotlin's standard library depends on the Java Class Library. [13]
The following is the UML Diagram for the class TapoSmartBulb. We reported only this diagram since the other ones are similar, and omitted to avoid repetition.

Figure 4.4: UML Class Diagram for Tapo Smart Bulb

Each class is equipped with 3 private attributes and some methods. The attributes manage a UiDevice element called device (immutable), an object model called smartObjModel (immutable), and an object state called smartObjState (mutable). Only the last one can be changed during the test execution since the state of the device must be updated depending on the action performed. An important assumption here is that the state of the device, otherwise defined will be switched on, and this is the rule that we decided to adopt. The methods, instead, allow us to perform actions, like moving among Views to reach the one in which there is the UI element that triggers the EVENT or directly triggers the EVENTs. Also, after generating an EVENT it is written the activity log (gtfile.txt) to record when the EVENT has occurred. We created a class for each app/smart device since the UI Elements and navigation of the View changes, only something is in common.

Then we created a utility file, named Utils.kt, that is not a pure Kotlin class, but it is a wrapper that contains utility components defined at package level, like enum classes, constant variables, or functions that are universal for our developments. The most important elements defined in this file are:

- `gtfile.txt` is the activity log file, that acts as ground truth, written by the writeGroundTruthFile function, containing the *EVENTs* just executed.
- `writeGroundTruthFile` function is in charge of filling in the gtfile.txt with the *EVENTs* just performed.
- `SMARTOBJ_EVENT_ITERS` immutable variable embodies an integer value related to the number of *EVENTs* to be generated during the test execution.

This class embodies variables, functions, and parameters common to all the smart object classes, such as the values of the UI Selectors extrapolated by UIAutomatorViewer.

Try-Catch (Exception Handling)

Sometimes happens that *EVENTs* was not registered and performed correctly by the cloud backend, this leads to a misalignment between the mobile app state and the true real state of the smart device, which could provoke a premature crash of the test being executed. We think that this issue could be caused by the app backend not being thought to a massive number of events in a short period (could potentially cause problems cloud-side, AWS in this particular case), and some kind of network latency.

To decrease the number of misalignments it has been introduced 2 delays. The first delay, named `DELAY_ACTION` was set empirically to 2 seconds, and took care of waiting for the *EVENTs* to be generated, before going on with the next task. We mean that the test

waits for the smart device to apply the change of state correctly before proceeding. The other one, named `DELAY_WINDOW` was set empirically to 5 seconds and took care of waiting for the UI Elements of the Android View to load before generating an EVENT. Those delays will not solve the problem entirely, sometimes a misalignment still happens, but combined with the next construct could allow the test to be executed correctly taking care of the issues that already could potentially occur.

The crash problem was almost overcome by introducing a try-catch pair. These statements were used to wrap the code that triggers the *EVENTS*,
which are the `device.click(pixel_x1,pixel_y2)` or `device.findObject(...).ACTION()` methods.

These statements are applied performing the reasoning as follows. We try to generate the EVENT, if the process succeeds the state changes and an EVENT record is filled in the activity log, otherwise an NOP record is filled out in the activity log, and nothing is performed, and the test proceeds. Those NOP were ruled out in the data analysis and discarded. The impact of those operations is very low. There were 116 NOP operations over 3000 *EVENTs* generated. Here is an example of how the try-catch was implemented.

```
try {
    device.findObject(...).click()

    // Success message
    writeGroundTruthFile(gtfile,
        "[TIMESTAMP: ${getTimestamp()}] "
                + "[EVENT COUNTER: ${SMARTOBJ_EVENT_NUMBER}] "
                + "[APP: ${obj.app.appName}] "
                + "[DEVICE TYPE: ${obj.dev.dev}] "
                + "[DEVICE MODEL: ${obj.mod}] "
                + "[ACTION: Edit color randomly]\n")

    setDelay(SmartObjDelays.DELAY_ACTION.delay)
    }
} catch (e: Exception) {

    // Error message
    writeGroundTruthFile(gtfile,
        "[TIMESTAMP: ${getTimestamp()}] "
                + "[EVENT COUNTER: ${SMARTOBJ_EVENT_NUMBER}] "
```

```
                + "[APP: ${obj.app.appName}] "
                + "[DEVICE TYPE: ${obj.dev.dev}] "
                + "[DEVICE MODEL: ${obj.mod}] "
                + "[ACTION: NOP - ${e.message.toString()
            .replace(",", "-")}]\n")
}
```

Another cause of test crashes is some kind of feedback popup that unpredictably showed up and messed the test up causing a crash.
This latter problem was handled by performing a check on the presence of those popups and making them disappear by clicking on the exit button. Here we show a snippet of pseudocode that performs this check.

```
// Method that handles the random feedback popups that appear on the
↪  current app view.
private fun checkPopUpFeedback() {

    // Check if the popup is on View Frame Layout.
    if (device.hasObject(By.text(
            SmartObjTextSelector.TAPO_FEEDBACK.textLabel))) {

        // Closing Popup window.
        device.findObject(By.res(
            SmartObjResourceId.TAPO_SMARTHOME_CLOSE_BTN.rid)).click()
    }
}
```

This last construct allowed us to handle crashes soundly.

Now that we understand how the logic of the test operates, it is important to understand how an Instrumented test is executed.

UI Automator: Test Execution Workflow

There are mainly 2 steps that are performed to run the test inside our emulator. The former is the building part in which our piece of code is transformed into an APK file, this task is carried out by the Gradle tool. The latter is the running part, in which the APK file is installed inside the emulator and the test is finally performed. This last phase could be done directly from Android Studio which does all the work for us, or it is possible to install the APK directly from the command line exploiting the ADB program.

We tried the two methods, and for our finalities, there is no difference, but when performance matters it is better to choose the second method using ADB since consumes less CPU.

What is Gradle? By starting the Instrumented Test's Gradle task(connectedAndroidTest), right after the build, two APKs are generated, and both are installed on the device. One of them is the APK of the built App, and the other one is an APK that has the Test code inside of it, which will be used by the Android Instrumentation process, to interact with the App.

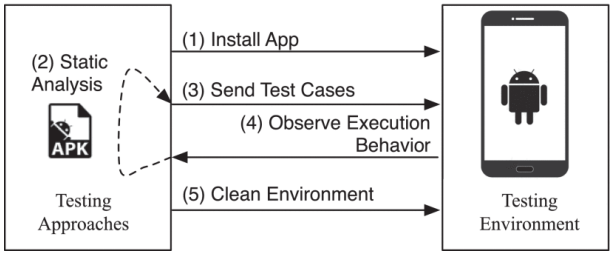

Figure 4.5: Android App Testing

What is an APK file? An APK is an archive file, meaning that it contains multiple files, plus some metadata about them.

What Are APK Files Used For? APK files allow you to install apps on your Android phone. When you open an APK on your device, it contains the instructions to install the app on your phone and provides information about the package itself to your device.

What is Android Debug Bridge (ADB)? Android Debug Bridge is a versatile command-line tool that lets you communicate with a device. [52] It is a client-server program that includes three components:

- A client, which sends commands. The client runs on your development machine. You can invoke a client from a command-line terminal by issuing an ADB command.
- A daemon (adbd), which runs commands on a device. The daemon runs as a background process on each device.
- A server, which manages communication between the client and the daemon. The server runs as a background process on your development machine.

How ADB works

When you start an ADB client, the client first checks whether there is an ADB server process already running. If there isn't, it starts the server process. When the server starts, it binds to local TCP port 5037 and listens for commands sent from ADB clients—all ADB clients use port 5037 to communicate with the ADB server. The server then sets up connections to all running devices. It locates emulators by scanning odd-numbered ports in the range 5555 to 5585, the range used by the first 16 emulators. Where the server finds an ADB daemon (adbd), it sets up a connection to that port. Note that each emulator uses a pair of sequential ports, an even-numbered port, for console connections and an odd-numbered port for ADB connections. Once the server has set up connections to all devices, you can use ADB commands to access those devices. Because the server manages connections to devices and handles commands from multiple ADB clients, you can control any device from any client (or from a script). ADB will be a fundamental tool to extract the ground truth file from the emulator.

4.2.5. Test Automation

The Automation is driven by the Kotlin class ExampleInstrumentedTest.kt. In this class we instantiated our smart devices objects, the UiDevice needed to interact with the emulated device touchscreen and then we define the loop `for (INTEGER in TOTAL_NUMBER) {...}` in charge of performing the automation work. Here we put an example for UiDevice and smart device class object instantiation.

```
val device: UiDevice =
↪  UiDevice.getInstance(InstrumentationRegistry.getInstrumentation())

// Object that is an instance of TapoSmartBulb.kt class L530E model.
val tapoSmartBulb = TapoSmartBulb(device=device,
↪  objState=SmartObjState.STATE_OFF)
```

The "for loop" is repeated for a total number of steps defined in the Utils.kt class in the immutable variable `SMARTOBJ_EVENT_ITERS`.

Performing some tests we estimated that 1440 events correspond to more or less 24 hours of the UI Instrumented Test running. With this estimation, we defined this variable as 3000 to be able to generate events for 2 days give or take. To simulate a real user we introduce a delay between 2 consecutive actions, named `DELAY_EVENT`, whose value is 60 seconds. This delay was also put in to be able to distinguish actions sharply and to have a clear view of what network packets are associated with an EVENT and what is associated

with spontaneous traffic.

The last thing worth mentioning is the seed variable, which is a random number between 1 and 4 that allows one to pick one of the smart devices at random to generate the next EVENT. We have adopted the following norm for the assignment of the seed:

1. is associated with the Tapo Smart Bulb L530E device
2. is associated with the EZVIZ Smart Bulb LB1 device
3. is associated with the Tapo Smart Plug P100 device
4. is associated with the EZVIZ Smart Plug T31 device

Below there is a snippet of this class.

```
for (i in 1..SMARTOBJ_EVENT_ITERS) {
    val st = System.currentTimeMillis()

    val seed = SecureRandom().nextInt(4).plus(1)

    when (seed) {
        1 -> tapoSmartBulb.selectRandomInstrumentedTest()
        2 -> EZVIZSmartBulb.selectRandomInstrumentedTest()
        3 -> tapoSmartPlug.selectRandomInstrumentedTest()
        4 -> EZVIZSmartPlug.selectRandomInstrumentedTest()
    }
}
```

4.2.6. Components

This section will present the specific devices used to validate the AutomIoT framework and subsequently collect the dataset. These are only possible examples of devices that could be managed with AutomIoT.
Concerning the general architecture illustrated in section 4.1, we have conceived the following test environment depicted in Figure 4.6, and Figure 4.7.

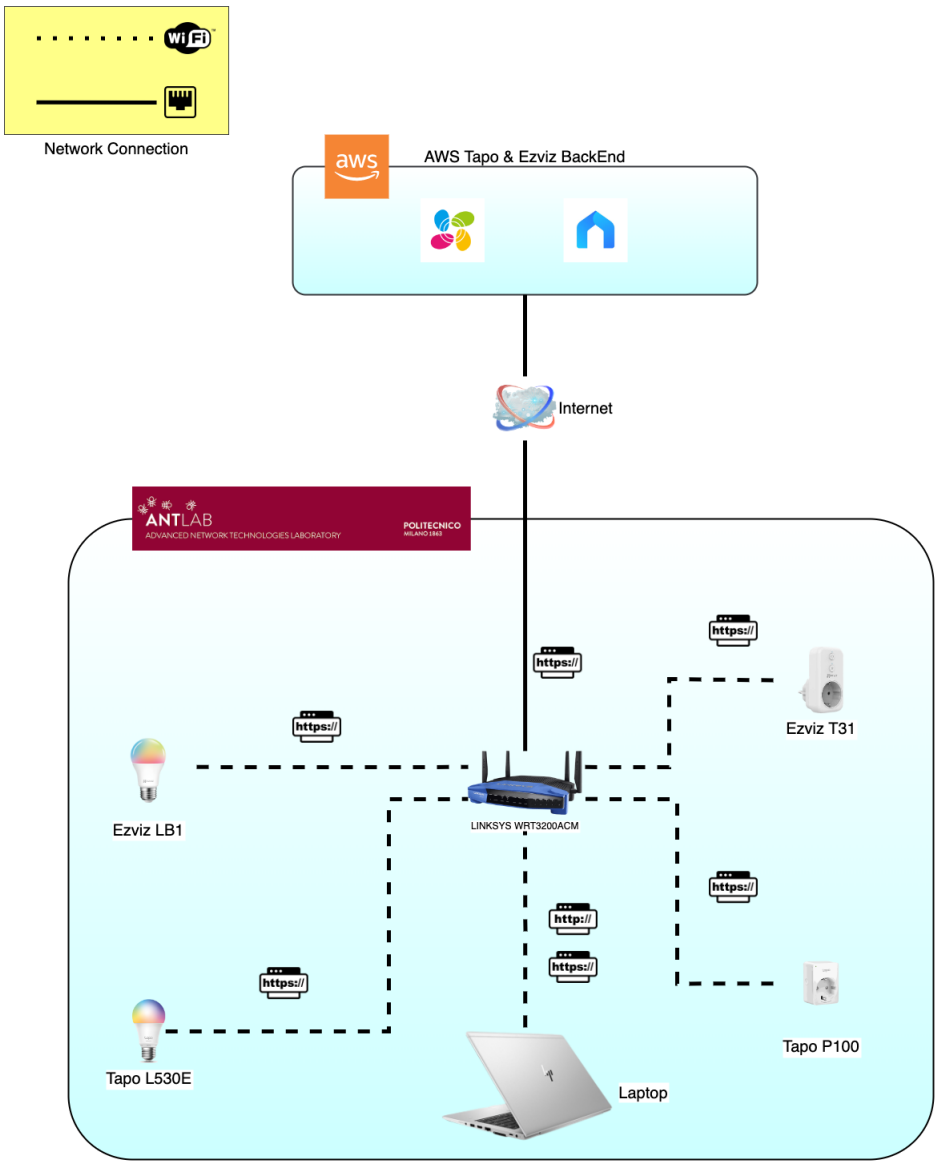

Figure 4.6: Test Architecture

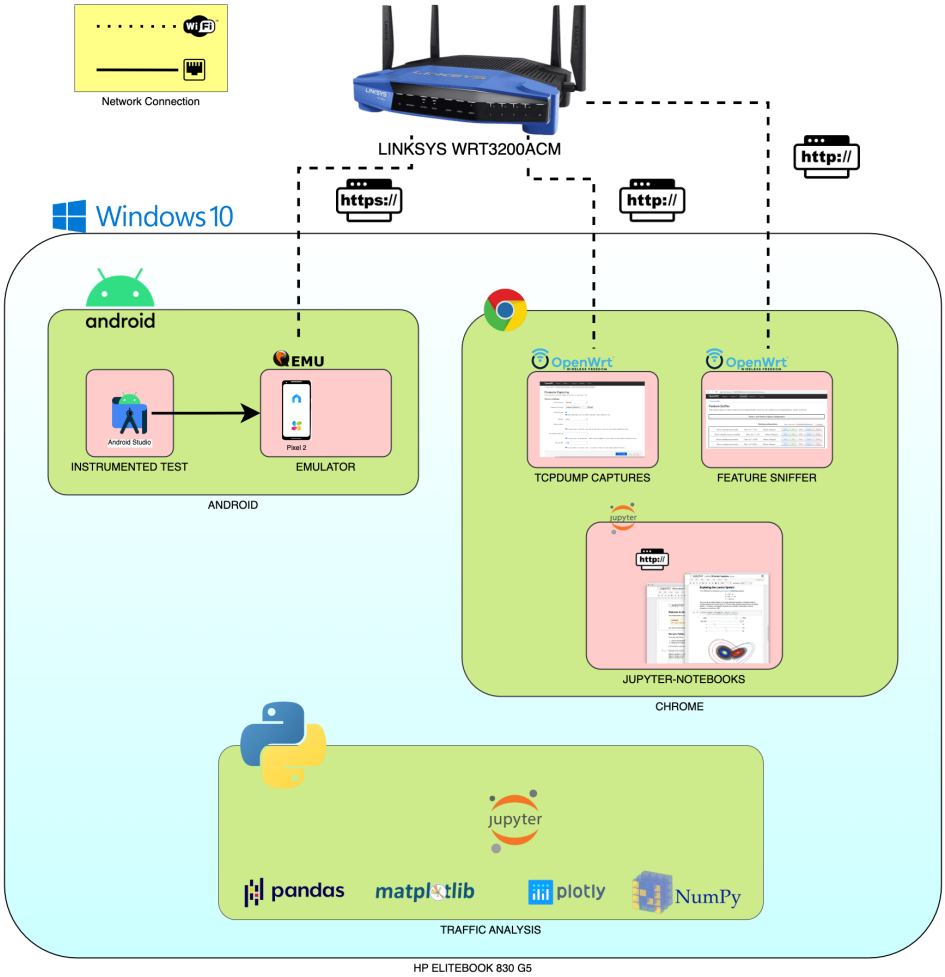

Figure 4.7: Laptop Overview

Smart home devices

The first element of our list is the Smart Device, an IOT device like a bulb or a plug that could be controlled via a mobile app. We have employed 2 smart plugs and 2 smart bulbs from 2 brands, EZVIZ and Tapo, that are shown in Figure 4.8. The firmware of the devices is proprietary software.

Computing Device

It's the device used for the Instrumented UI Test development, test, and execution. In the early stages, this process was done on a MacBook. Then, finally, after the UI Test has been stable and does not produce any more software problems to fix, it has been used

(a) EZVIZ LB1 [32]

(b) Tapo L530E [19]

(c) EZVIZ T31 [33]

(d) Tapo P100 [20]

Figure 4.8: Smart home devices

the HP EliteBook to run the test while collecting the network traffic. We can view the Macbook as a development/testing environment, where we have tested and conceived our solution, and the EliteBook as a production environment. These devices have been used also to run the emulated mobile device, instead the MacBook was used also for the data analysis.
Below are provided the hardware specifications of the two computing devices exploited.

- MacBook Pro Early 2015: Laptop distributed by Apple with a 2.7GHz dual-core Intel Core i5 processor (Turbo Boost up to 3.1GHz) with 3MB shared L3 cache, 128 GB SSD, 8 GB RAM, and macOS Monterrey 12.6.4 [30]
- HP EliteBook 830 G5: Laptop distributed by HP with a i5 8250U 1.6GHz CPU, 8 GB RAM and Windows 10 OS. It is provided by ANTLab laboratory [16].

Mobile Device and Apps

It has been emulated by exploiting Android Studio developing tools. The emulated device runs inside the laptop. We installed the 2 management apps needed for our purposes that are Tapo (distributed by Tapo, a brand of TP-Link Technologies Co., Ltd. [18]) and EZVIZ (distributed by EZVIZ, a brand of Hikvision - Hangzhou Hikvision Digital Technology Co., Ltd.) [31].

Post-data processor

This device is only the Macbook, used to extract the data that we needed to get an answer to our questions for this work.

Network Packet Sniffer

This device has been used as an Access point, as a Router, and as a network packet sniffer.

Linksys WRT3200ACM: is an MU-MIMO Gigabit Wi-Fi Router with a dual-core CPU and 512MB RAM, installed in the ANTLab laboratory [42], [43]. It has been used to sniff the network traffic generated between the smart device and the Android-emulated device.

Figure 4.9: Linksys WRT3200ACM

OpenWrt The AP utilizes OpenWrt, a Linux operating system targeting embedded devices, with network tools to be set as a local modem/router in a LAN. [61], [62]

Python Feature Sniffer The AP has been customized with a Python3-based tool able to extrapolate statistics on a specific device, filtering it by its MAC address, on a time base with windows of X seconds. It was called Feature Sniffer. Feature Sniffer is a tool for capturing traffic features on the fly in an Access Point with OpenWrt firmware. [63] [83]. The following are examples of how it is represented in the GUI of the tool.

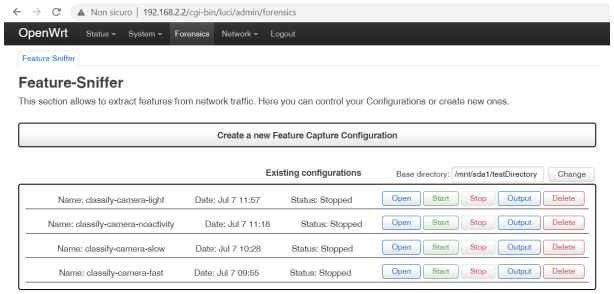

Figure 4.10: Feature Sniffer Home page

Figure 4.10 depicts the homepage of the tool from which it could be possible to create a new feature capture configuration, while figure 4.11 illustrates the details of a parameters setup.

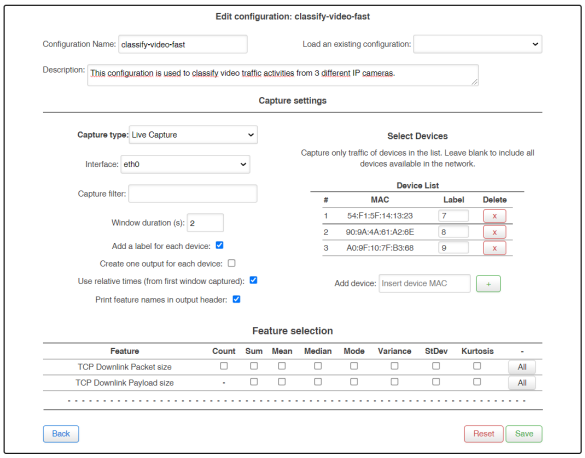

Figure 4.11: Feature Sniffer Aggregation config

It is possible to filter by device MAC address, define a capture window duration, and select the interface, and choose possible features related to the packets collected.

4.2.7. Data Gathering

This phase is based on the activation of 3 elements. We had to plug in all the smart devices to the power source, then it was needed to switch off all the smart devices, and afterward, we had to enable the Tcpdump feature of the sniffer from its Web page. Finally, we could run the test from Android Studio. As already mentioned during the test execution the "gtfile.txt" is filled in with *EVENTs* just generated. For the test, the smart device and the variable inside the `ExampleInstrumentedTest.kt` was initialized as `STATE_OFF`, so switched off. The entire test lasted 2 days and 3 hours approximately for about 3000 EVENTS generated. The test was run during the weekend in the ANTLab laboratory. The NOP operations were 116 over 3000 *EVENTs* (3.87%).

After the test had finished, we disabled the Tpcump feature of the sniffer, we connected using SSH protocol to the filesystem of the Linksys router and pulled from it the network traces. There were 4 of them, named capture.pcap, capture1.pcap, capture3.pcap, capture4.pcap. The evidence of the duration was extracted by inspecting the 4 network traces. Here we share the main features.

```
File name:           capture.pcap
Number of packets:   3,057 k
File size:           2 GB
Capture duration:    69072.681900 seconds (about 19 h)
First packet time:   2022-09-20 17:19:24.476804
```

```
Last packet time:    2022-09-21 12:30:37.158704

File name:           capture1.pcap
Number of packets:   2,868 k
File size:           2 GB
Capture duration:    90654.061987 seconds (about 25 h)
First packet time:   2022-09-21 12:30:37.172284
Last packet time:    2022-09-22 13:41:31.234271

File name:           capture2.pcap
Number of packets:   2,721 k
File size:           2 GB
Capture duration:    14642.690737 seconds (about 4 h)
First packet time:   2022-09-22 13:41:31.234278
Last packet time:    2022-09-22 17:45:33.925015

File name:           capture3.pcap
Number of packets:   680 k
File size:           583 MB
Capture duration:    9648.013583 seconds (about 2.5 h)
First packet time:   2022-09-22 17:45:33.925034
Last packet time:    2022-09-22 20:26:21.938617
```

We exploited the ADB tool to extract the "gtfile.txt" from the emulator.

4.2.8. Data Analysis

We performed a twofold kind of analysis, we created Time series trying to understand how the network traffic changes regarding the *EVENTs* generation, and we created Sankey diagrams to discover how many packets were associated with EVENTS and spontaneous traffic, what IPs/services or ports were used.

In this context, when we use the term *"data"*, it means Pcap and CSV files containing network traffic data and ground truth events generated by the emulated mobile device. The Pcap files are files written in a specific format that contain network traffic records.

Before starting the analysis with the Python libraries, we performed some kind of pre-inspection by adopting the command line tool Tshark offered by the Wireshark tool. Through this pre-inspection, we were able to understand that the backend of Tapo and

EZVIZ were hosted using AWS services. Here we put 2 examples of the one that we encountered.

```
## TAPO TP-LINK
> nslookup tapo.tplinkcloud.com

Non-authoritative answer:
tapo.tplinkcloud.com        canonical name =
↪  d3utcvynanh8w7.cloudfront.net.
Name:        d3utcvynanh8w7.cloudfront.net
Address: 108.139.243.127
Name:        d3utcvynanh8w7.cloudfront.net
Address: 108.139.243.11
Name:        d3utcvynanh8w7.cloudfront.net
Address: 108.139.243.108
Name:        d3utcvynanh8w7.cloudfront.net
Address: 108.139.243.120

## EZVIZ
> nslookup euauth.EZVIZlife.com

Non-authoritative answer:
euauth.EZVIZlife.com        canonical name =
↪  EZVIZlife-auth-16461422.eu-west-1.elb.amazonaws.com.
Name:        EZVIZlife-auth-16461422.eu-west-1.elb.amazonaws.com
Address: 54.228.215.71
Name:        EZVIZlife-auth-16461422.eu-west-1.elb.amazonaws.com
Address: 34.241.142.38
```

TShark [85], provided by Wireshark [86] v3.6.8, is a tool in charge of reading Pcap files, manipulating them, and extrapolating only the needed information, ruling out the redundant ones. Along with those FQDNs that characterized AWS cloud resources, there are also in the network packets FQDNs that refer to web services not strictly connected with Tapo and EZVIZ ones. For example, we found FQDNs related to Facebook, TikTok, Google, Snapchat, and so on. They generate a significant amount of network traffic, and we could guess that this is related to profiling user behavior, for data analytics, or other purposes.

The data analysis was performed using Python 3 (v.3.7.3) programming language, and in particular, exploiting the following libraries:

- Jupyter notebook (v.6.4.12) for code editing (data transformation and data visualization). [38]
- NumPy (v.1.21.6) for data transformation. [51]
- Pandas (v.1.3.5) for data transformation. [64]
- Plotly (v.5.10.0) for Sankey Diagrams. [65]
- Matplotlib (v.3.5.2) for Time Series. [78]

The notebook is essentially a web application accessed through a browser that is used for our developments. The source code of the notebook is available at the following links [66], [67], [68], [69]. The following is an example of how the Web page is presented:

Analisi trace di rete su dispositivi

Smart Device Network Traffic (SDNT)

SMARTBULB EZVIZ 64:F2:FB:DF:FB:E1 -> LABEL 1

~~SMARTPLUG EZVIZ 64:F2:FB:48:2C:5B -> LABEL 2~~

SMARTBULB TAPO 00:5F:67:BF:09:EF -> LABEL 3

SMARTPLUG TAPO E8:48:B8:D6:A8:1D -> LABEL 4

In [ ]:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import time
from datetime import datetime as dt, timedelta as td
import time
import plotly.express as px
import socket
import glob
import os
```

In [ ]:

```
#
# LABELS DEFINITION
# Device are identified by MAC address associated to the device by the linksys router
# during the tcpdump capturing phase
#
# Ezviz Smart Bulb LB1
label1 = {"mac":"64:F2:FB:DF:FB:E1","app":"EZVIZ","device":"Smart Bulb","name":"LB1"}
# Ezviz Smart Plug T31 (device capture data non available since not connected to same access point, but to other Wi-Fi)
# label2 = {"mac":"64:F2:FB:48:2C:5B","app":"EZVIZ","device":"Smart Plug","name":"T31"}
# Tapo Smart Bulb L530E
label3 = {"mac":"00:5F:67:BF:09:EF","app":"Tapo","device":"Smart Bulb","name":"L530E"}
# Tapo Smart Plug P100
label4 = {"mac":"E8:48:B8:D6:A8:1D","app":"Tapo","device":"Smart Plug","name":"P100"}
```

Figure 4.12: Jupyter Notebook Example

The last thing that we want to point out is that one of the devices was ruled out from the

analysis since we decided to design this part to perform a benchmark between the same type of device for different brands

`Tapo SmartBulb vs EZVIZ SmartBulb`

and between different types of devices for the same brand.

`Tapo SmartBulb vs Tapo SmartPlug.`

The device excluded from the analysis was the EZVIZ T31 Smart Plug.

Time Series

This analysis was based on 2 data sources, the gtfile.txt with the *EVENTs* recorded and the capture01000ms_seg0104.csv, capture01000ms_seg0204.csv, capture01000ms_seg0304.csv, capture01000ms_seg0404.csv files. Those last CSV files are the ones obtained by applying an aggregation window offered by the Python feature Sniffer tool extracting specific features from the 4 Pcap files. The feature in our case was a temporal window of 1 second in which packets are aggregated filtering by MAC addresses of smart devices. This feature does not take into account the packet payload, but only the overhead. We have also performed a transformation for what concerns the ground truth file. The structure of the activity log was originally collected in the following format:

```
----------------------
gtfile.txt row example
----------------------
[TIMESTAMP: 2022-09-22 19:15:45.740] [EVENT COUNTER: 2985] [APP: EZVIZ]
↪  [DEVICE: Smart Bulb] [ACTION: Edit color temperature randomly]
```

We have converted the Txt file more concise CSV file, for analysis purposes, containing the following information:

```
----------------------
gtfile.csv row example
----------------------
TIMESTAMP,EVENT COUNTER,APP,DEVICE,ACTION
2022-09-20 17:31:20.935,1,Tapo,Smart Bulb,Turn ON bulb
```

By exploiting a bar graph we have plotted the data from those CSV sources. Each graph has a time window of 1 hour for a meaningful figure that could be analyzed correctly. On the x-axis, we have timestamps using the format `"%Y-%m-%d %H:%M:%S.%f"` for instance (e.g. `"2022-09-22 19:15:45.740"`). On the y-axis, we have the packet size measured

in Bytes. The blue bars are associated with network packets collected with the AP, and the red bars refer to *EVENTs*. The length of the bar is equivalent to the packet size in that particular timestamp. Here we put an example of a time series produced, which will commented on in detail in the next chapter.

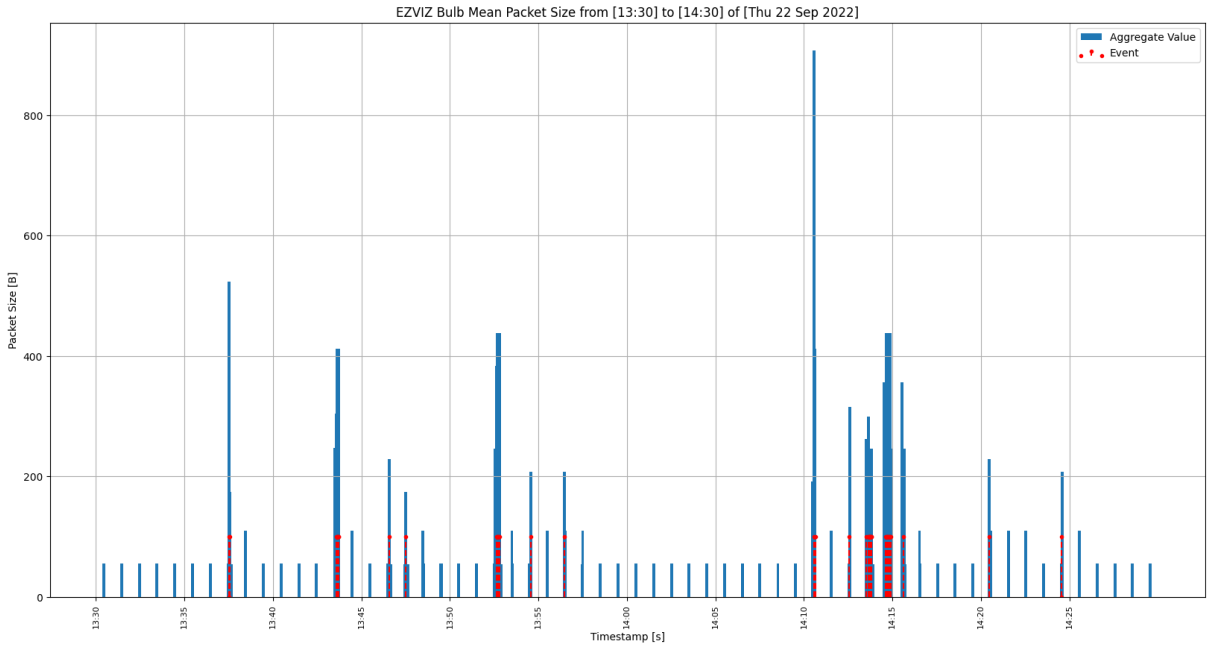

Figure 4.13: Time Series Example

Sankey Diagrams

Those diagrams are created by transforming directly the 4 Pcap files extracted from Linksys. We have exploited Tshark to produce a unique CSV file, named sankey.csv. We filtered the network data by MAC addresses associated with the Smart home devices. The Tshark command used is the following:

```
tshark -r capture.pcap -t ud -Y 'eth.addr==64:f2:fb:df:fb:e1 or
↪  eth.addr==64:f2:fb:48:2c:5b or eth.addr==00:5f:67:bf:09:ef or
↪  eth.addr==e8:48:b8:d6:a8:1d' -T fields -e _ws.col.Time -e eth.src -e
↪  ip.src -e udp.srcport -e tcp.srcport -e eth.dst -e ip.dst -e
↪  udp.dstport -e tcp.dstport -e _ws.col.Protocol -e frame.len -E
↪  separator=, -E occurrence=f 2>/dev/null > sankey.csv
```

That produced a CSV file with these fields:

```
TIMESTAMP
MAC_SRC
```

IP_SRC
UDP_SRC
TCP_SRC
MAC_DST
IP_DST
UDP_DST
TCP_DST
TCP/UDP PROTOCOL
PACKET_SIZE

To create the Sankey Diagrams we took hints from the one provided on the Web page of IOT Analytics of UNSW Sydney [81]
We have produced distinct Sankey diagrams depending on the flow of the packet, so we have a Sankey diagram for network packets that have as source the smart device MAC address, named OUTGOING, and a second one for network packets that have as destinations the smart device MAC address, named INCOMING. Here we put an example of a Sankey diagram produced (OUTGOING in this case), which will commented on in detail in the next chapter.

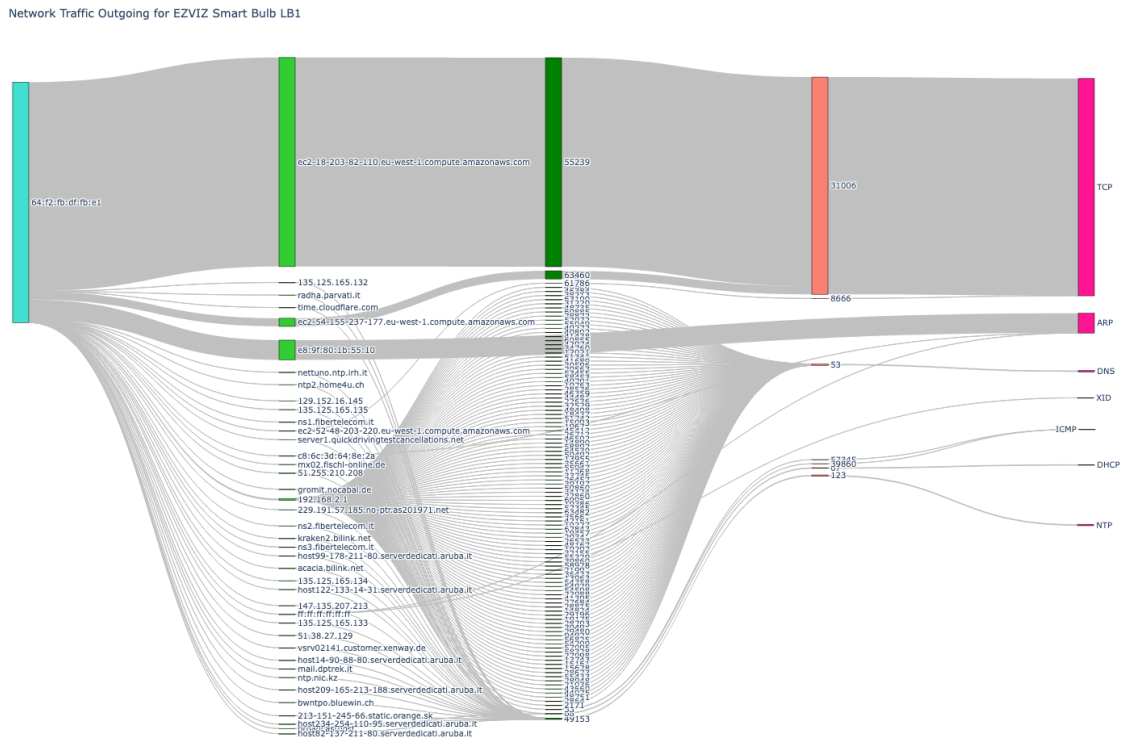

Figure 4.14: Sankey Diagram Example

With these diagrams, we would like to point out the network traffic exchanged. We have 5 short and long pillars/rectangles highlighted with different colors.
They are from left to right

SOURCE SMART DEVICE MAC ADDRESS
DESTINATION FQDN/IP/MAC ADDRESS
TCP/UDP SOURCE PORT
TCP/UDP DESTINATION PORT
TCP/UDP PROTOCOL

for the OUTGOING packets, and

SOURCE FQDN/IP/MAC ADDRESS
DESTINATION SMART DEVICE MAC AD
TCP/UDP SOURCE PORT
TCP/UDP DESTINATION PORT
TCP/UDP PROTOCOL

for the INCOMING packets.
In the middle, in grey, we have the network traffic exchanged.

5 | Experimental Results

In this chapter, it will be presented the main results collected from the data-gathering step. Essentially, we have performed 2 types of analysis. The first one is based upon a Time series in which we have looked for any kind of hints to discern Spontaneous Network Traffic (SNT) from EVENTs Network Traffic (ENT) for a specific smart device. Then, we analyzed possible contrasts in Sankey diagrams from packets sent (outgoing flow) and received (incoming flow) from the smart device. We made a benchmark between the Tapo smart bulb and the EZVIZ smart bulb, so different brands for the same type of device, and then between Tapo smart bulb and Tapo smart plug, so same brand and different device type. Finally, we have produced and differentiated the two flows of traffic. It will be shown that spontaneous traffic, particularly for the EZVIZ brand produced a lot of network traffic compared to the Tapo brand.

5.1. Time series Analysis

In this section, we will explain the main choice made for the time window parameter, and discern the two flows of network data.

5.1.1. Aggregation capture tuning

We have performed different kinds of aggregation (0.5s, 1s, 5s, and 10s) with the Python Feature Sniffer tool offered by the AP. We have selected the one that best fits our purposes, that is the window of 1 second. Here we provide a graphical example for each one of them.

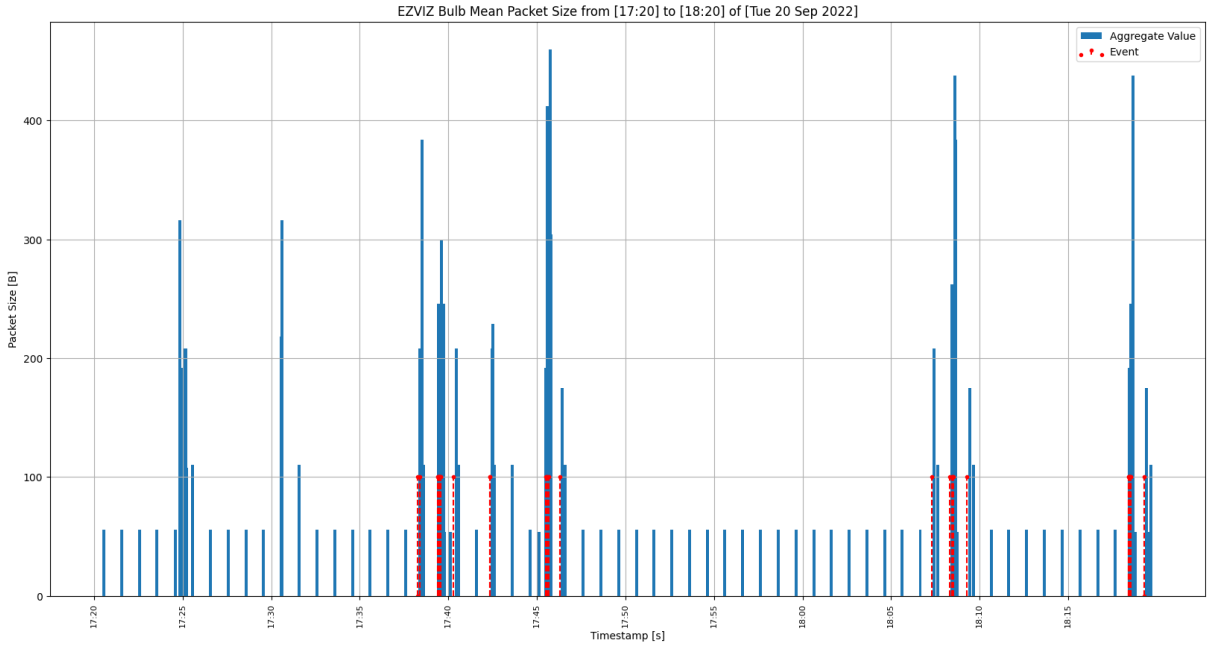

Figure 5.1: Time series windows 0.5s

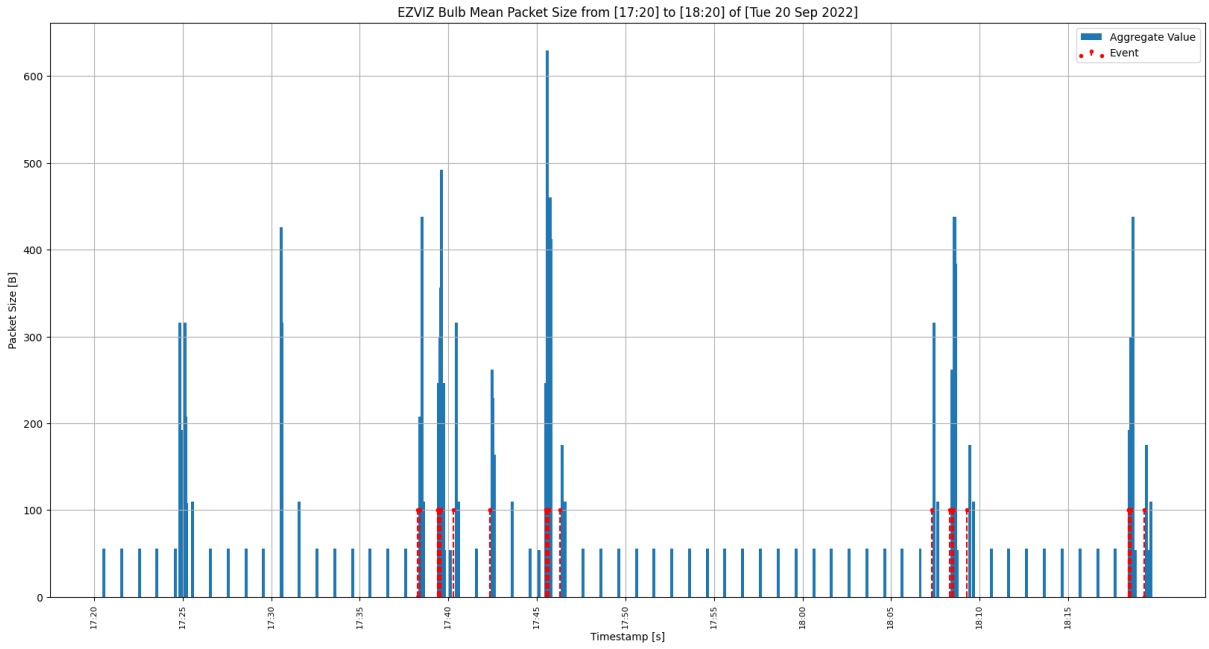

Figure 5.2: Time series windows 1s

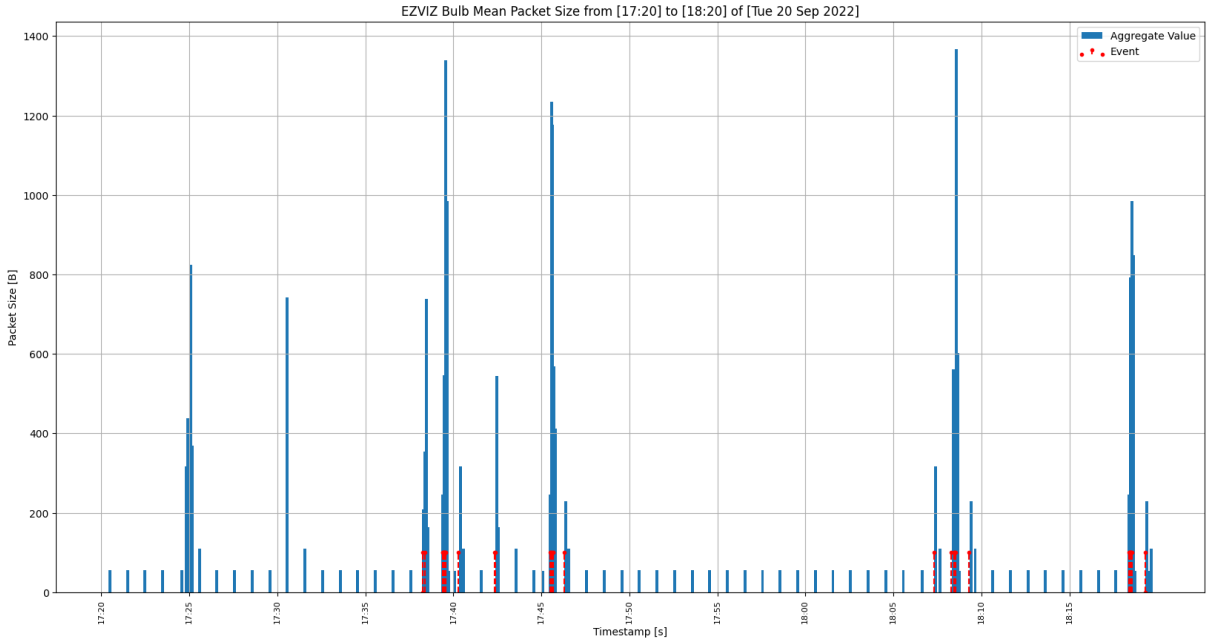

Figure 5.3: Time series windows 5s

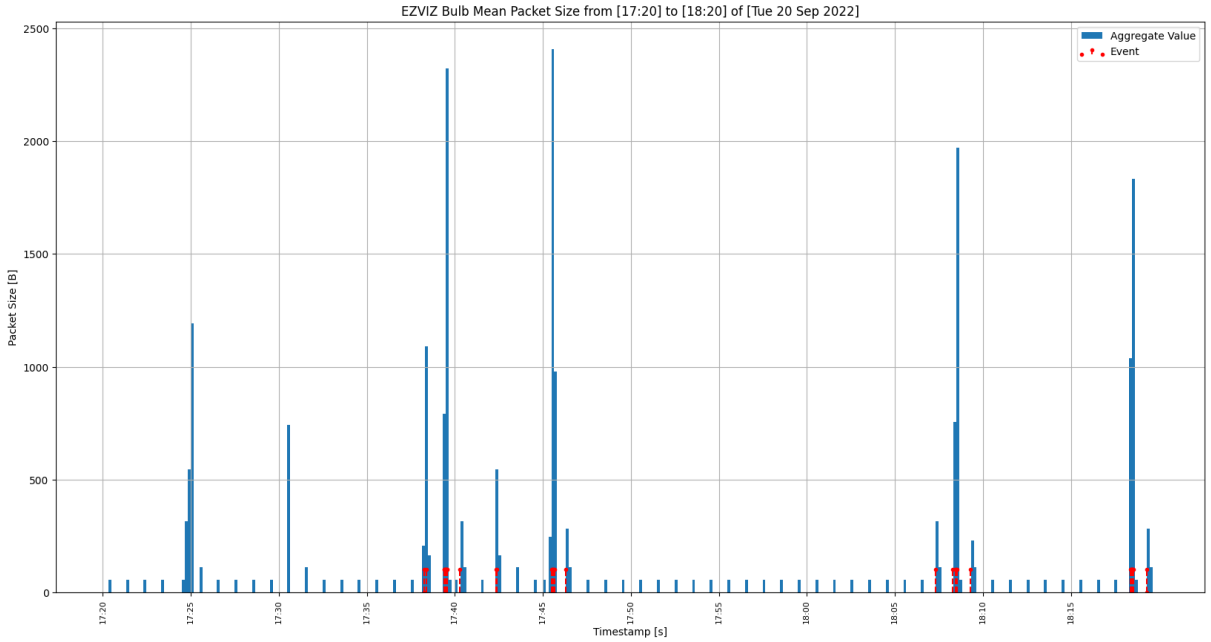

Figure 5.4: Time series windows 10s

As we can see, the choice of 1 second for aggregating the packets was driven to find a good compromise. Choosing 10s would have meant an inferior plot qualitatively and less readable.

5.1.2. Example capture Day 1 (20-09-2022)

We have reported 3 examples, one for each day of data gathered regarding the LB1 EZVIZ Smart Bulb device. The behavior of the Time series of the other 2 devices involved in the benchmark was identical and will be omitted. To discern ENT and SNT it is used a time window of 2 seconds. If we pick the timestamp of an EVENT from the activity log, we create a time window of 2 seconds around it by subtracting (lower bound timestamp) and adding (upper bound timestamp) an offset of 1 second. This must not be confused with the aggregation time window that has been mentioned before. This is another window used to discriminate what packets are considered associated with EVENTs traffic, and what packets are associated with spontaneous traffic. The traffic contained within those windows has been considered EVENTs traffic. The rest has been considered spontaneous. This value has been chosen considering the delay used in the EVENTs generation phase, described in section 4.2.4 of chapter 4. In particular, considering the value associated with the `DELAY_ACTION)` set to 2 seconds. Here, the blue bars distinguished all the network packets generated. The ones in red instead described the timestamp in which an EVENT occurred. As we can see, corresponding to those red bars, we have a spike, so we can discern spontaneous from EVENT traffic easily with a linear function that defines a bound over which if are found packets, then we have an EVENT; otherwise, the traffic is considered as spontaneous. As already said, this could be defined as 150 Bytes. This is the first example that shows the first-day capture.

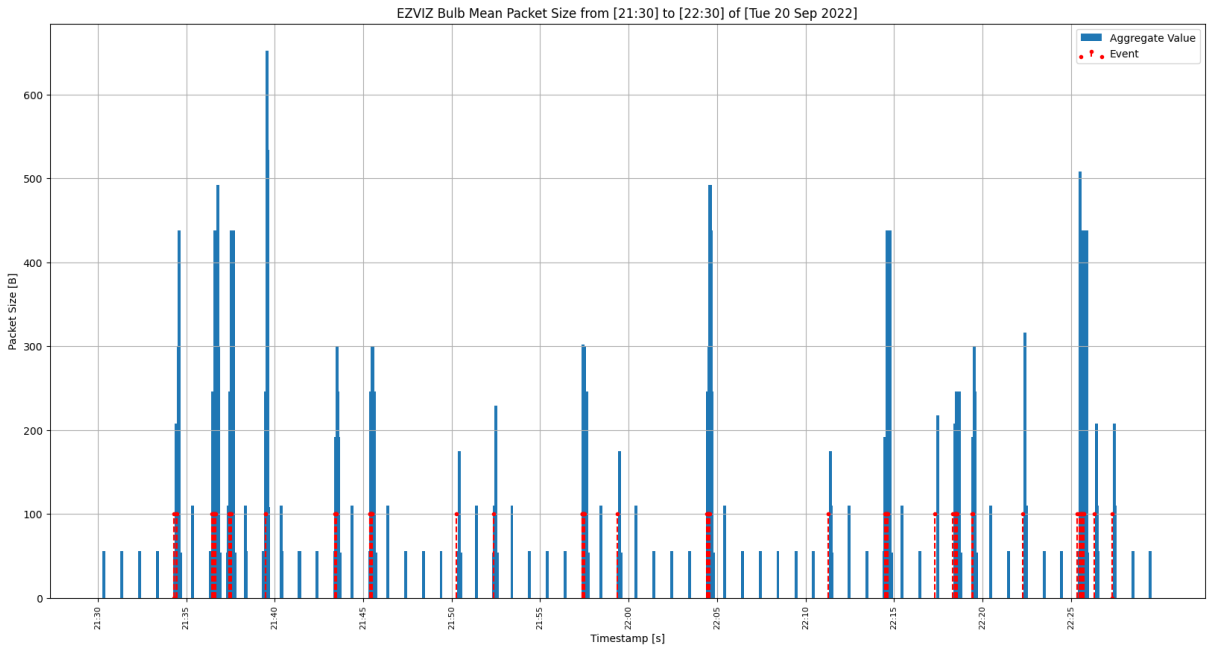

Figure 5.5: Time series EZVIZ LB1 of 20-09-2022 from 21:30:00 to 22:30:00

5.1.3. Example capture 2nd day (21-09-2022)

This is an example that reports the second-day capture.

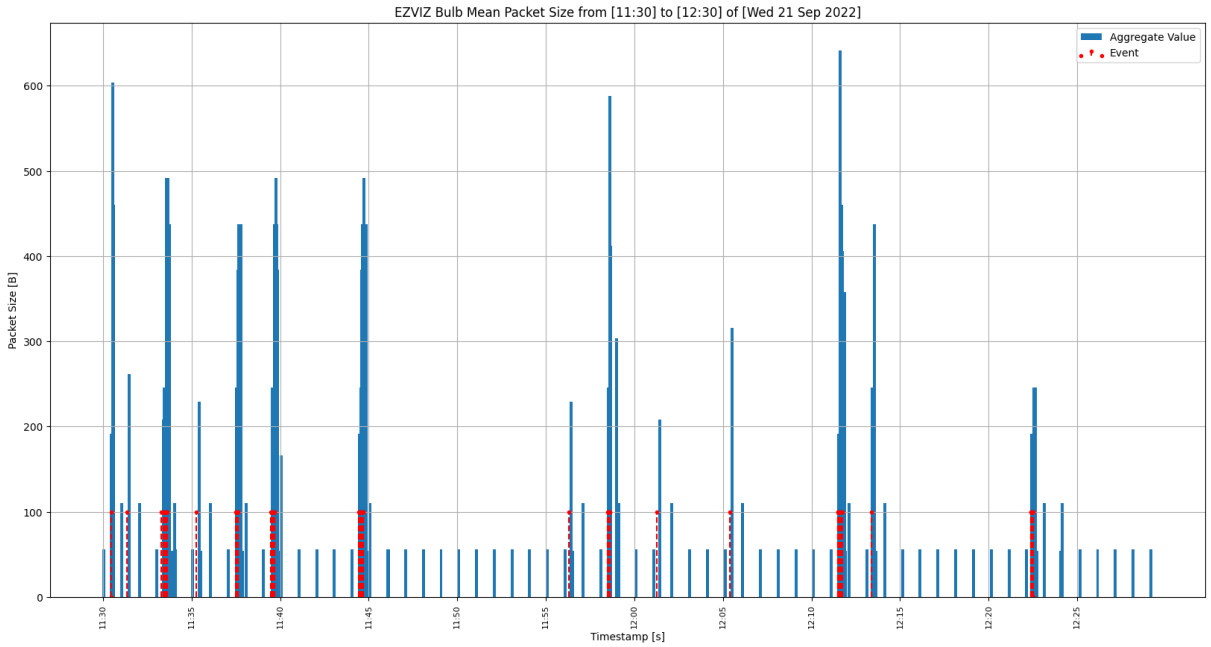

Figure 5.6: Time series EZVIZ LB1 of 21-09-2022 from 11:30:00 to 12:30:00

5.1.4. Example capture 3rd day (22-09-2022)

This is the last example that reports the third-day capture.

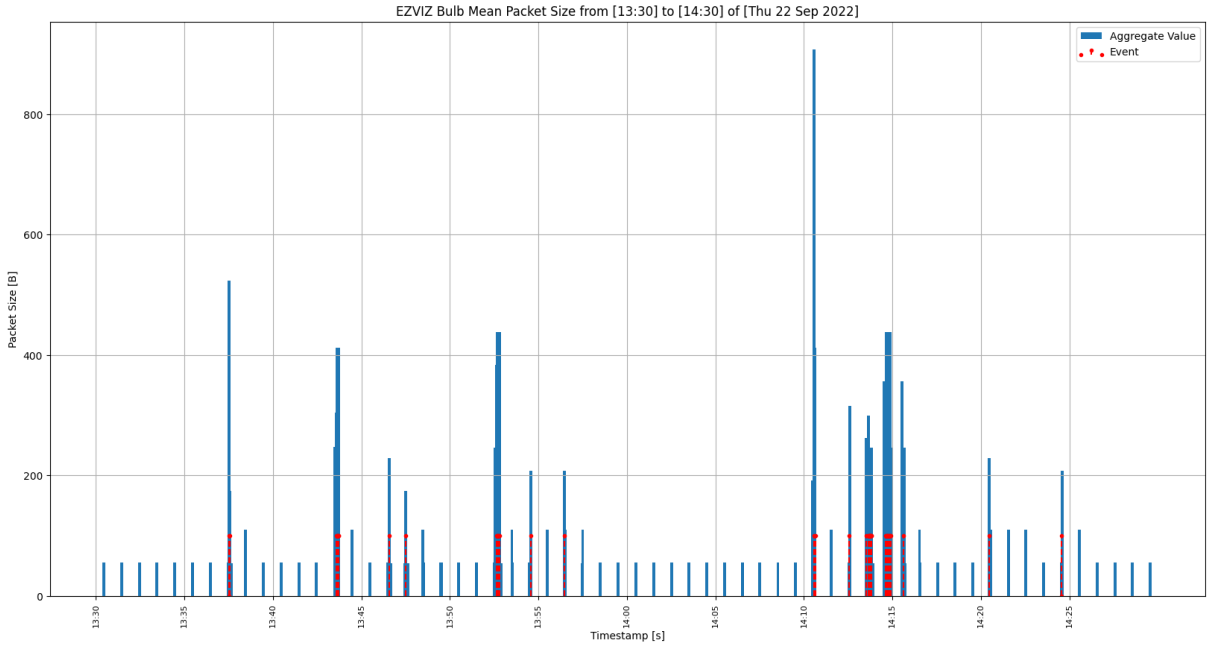

Figure 5.7: Time series EZVIZ LB1 of 22-09-2022 from 13:30:00 to 14:30:00

This tendency that we have commented on and seen in the previous plots does not change over time, indeed we saw 3 examples with similar behavior, and the proximity of EVENTs that spike in the network traffic increase, and far from them, there are some network packets unrelated that could be associated with network traffic. The next chapter will give a quantitative measure of spontaneous traffic.

5.2. Sankey diagram Analysis

This section elucidates the main qualitative comparisons performed using Sankey diagrams. We tried to find differences between outgoing and incoming traffic for the same device; when we have the same type of device but different brands; when we have different types of devices for the same brand and finally which is the impact of a smart device EVENTs and spontaneous traffic diagram over the overall dataset, also by giving a quantitative measure.

5.2.1. Outgoing - Incoming traffic

Looking at the two flows of traffic for Tapo L530E, taken as reference object arbitrary, depicted in Figure 5.8 5.9, we could spot that are mostly identical.

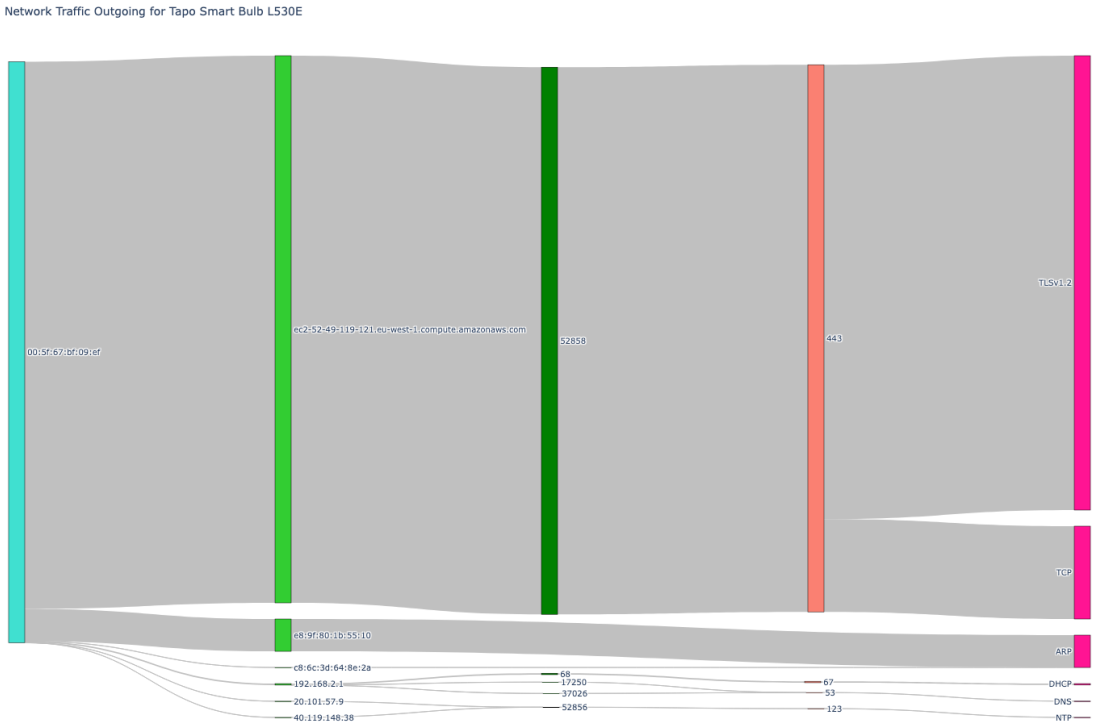

Figure 5.8: Sankey diagram All Traffic Tapo L530E Outgoing

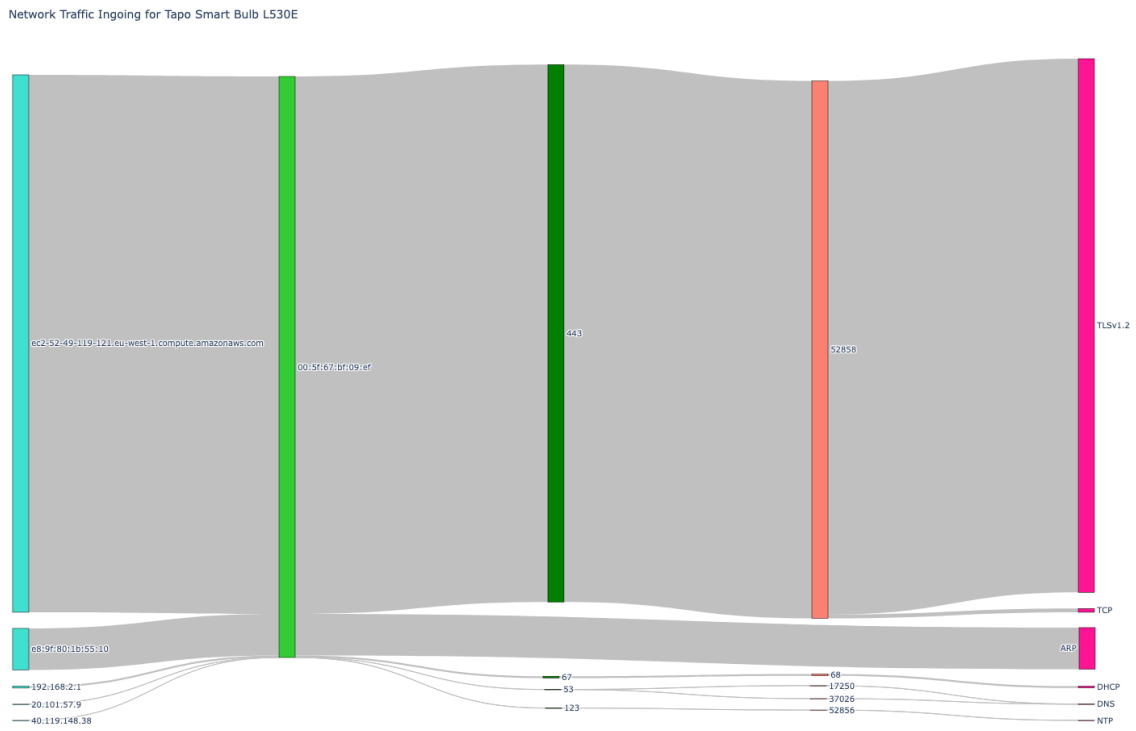

Figure 5.9: Sankey diagram All Traffic Tapo L530E Incoming

The main aspect to consider is the usage of AWS web services to host the backend of the smartphone applications both for Tapo and EZVIZ. Indeed, there is no direct interaction between the device and the controlling application. The flow is brokered by the AWS backend. Almost the total of the packets sent or received by the smart device goes through the cloud services.

5.2.2. Tapo bulb vs EZVIZ bulb

Figure 5.11 and Figure 5.10 show the OUTGOING Sankey for L530E and LB1. EZVIZ app produced a lot of DNS and NTP traffic with respect to the Tapo counterpart. There is a lot of DNS, NTP traffic, and noise traffic (traffic from different sources, and websites that one does not expect to be there), and may be related to the integration of external services, or third-party analytics services as spotted for instance by [47], [47].

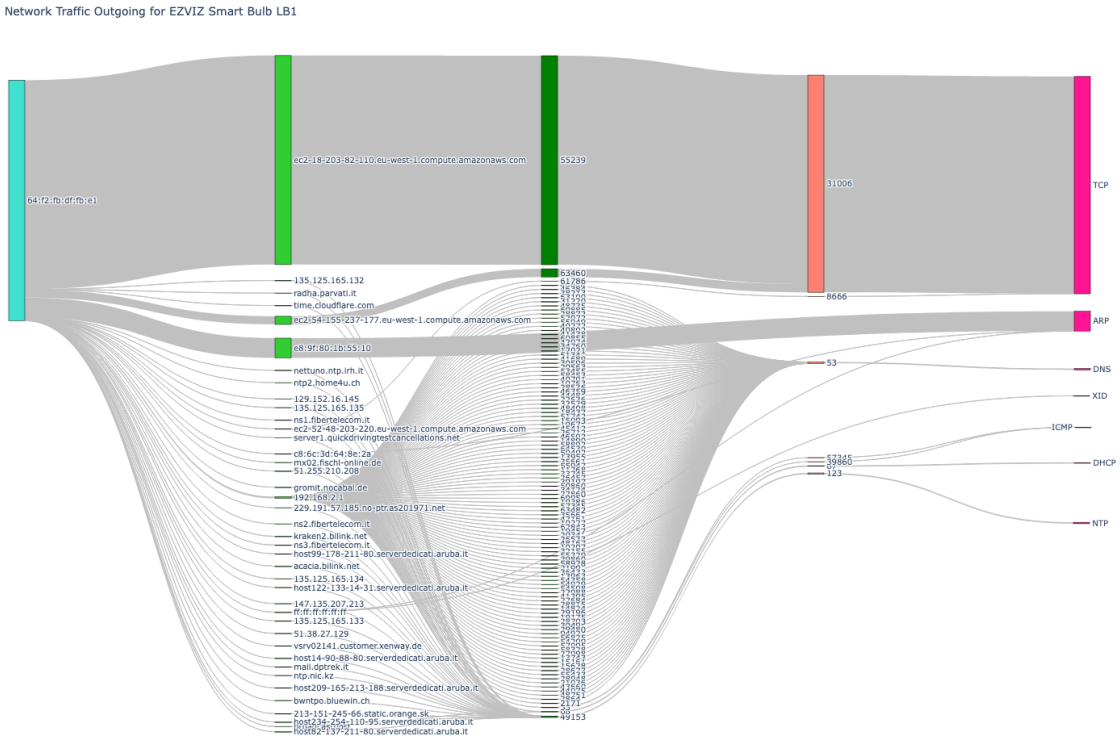

Figure 5.10: Sankey diagram All Traffic EZVIZ LB1 Outgoing

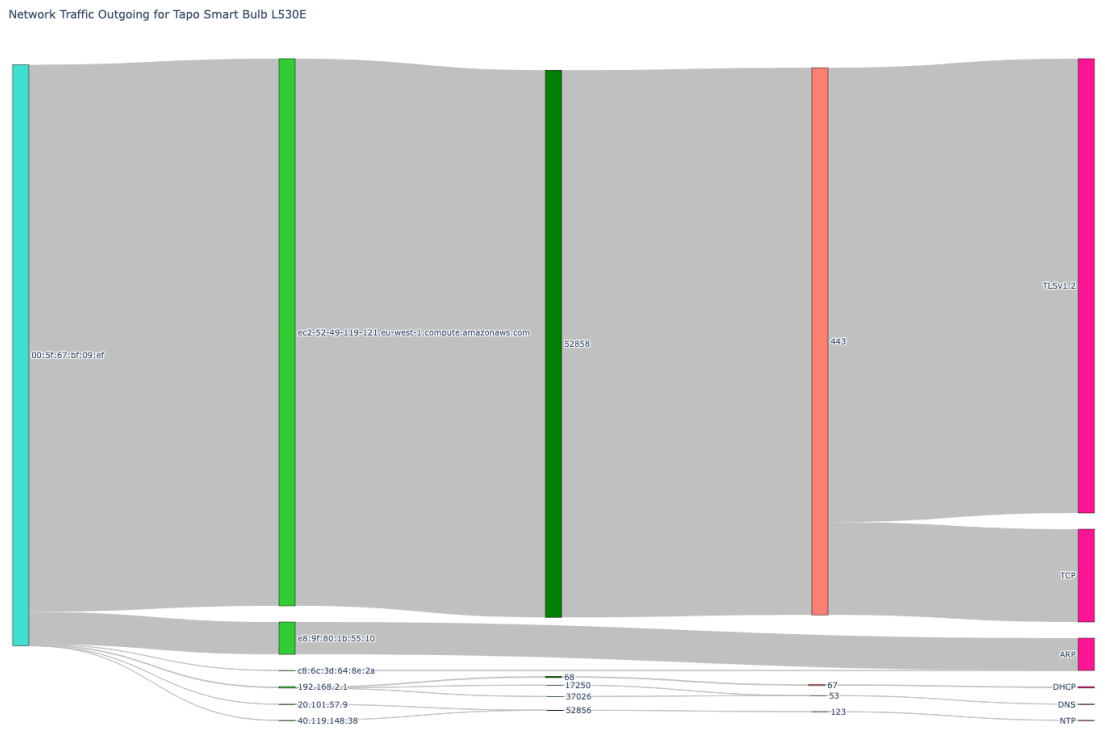

Figure 5.11: Sankey diagram All Traffic Tapo L530E Outgoing

5.2.3. Tapo bulb vs Tapo plug

Different devices from the same brand do not show a significant change, they are very similar.

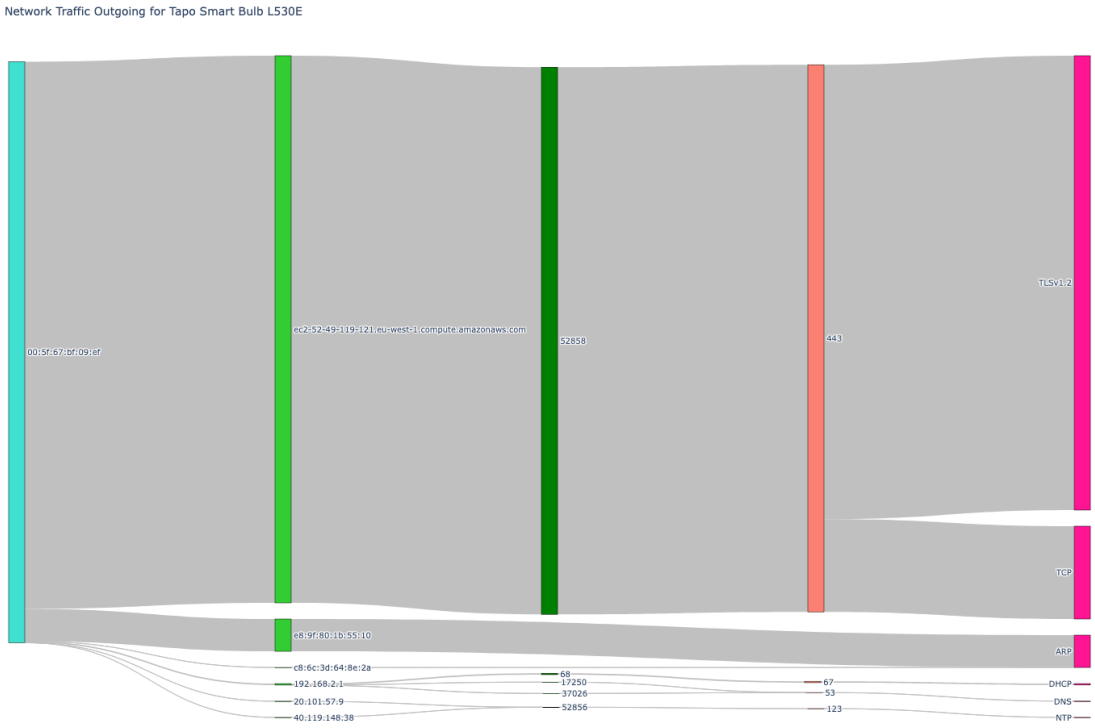

Figure 5.12: Sankey diagram All Traffic Tapo L530E Outgoing

Figure 5.12 and Figure 5.13 show the OUTGOING Sankey for L530E and P100.

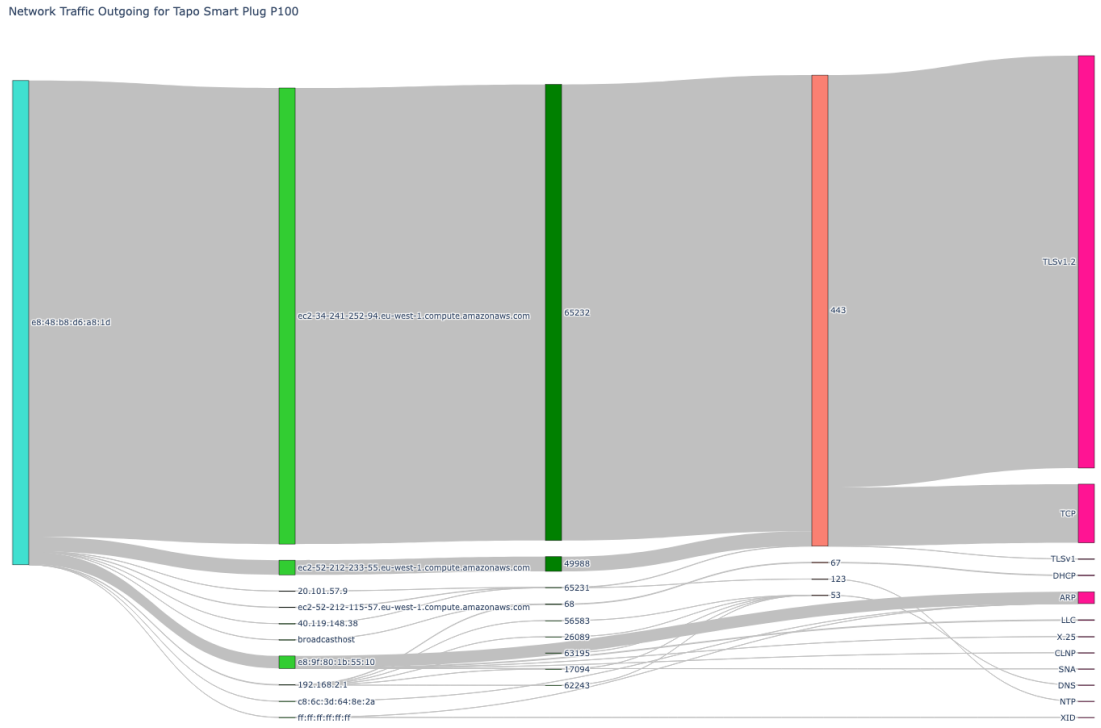

Figure 5.13: Sankey diagram All Traffic Tapo P100 Outgoing

5.2.4. All traffic vs EVENTs vs spontaneous traffic

Figure 5.14, 5.15 and 5.16 show respectively the total OUTGOIG network traffic, the EVENTs network traffic and spontaneous network traffic generated by EZVIZ LB1 device.

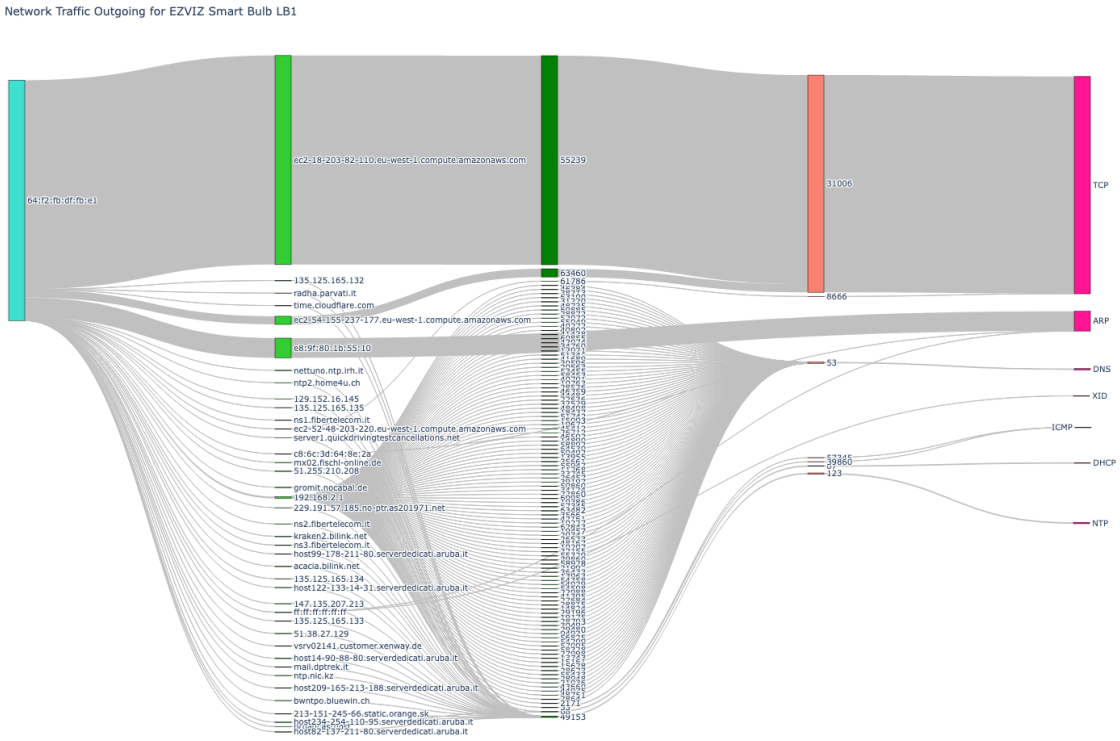

Figure 5.14: Sankey diagram All Traffic EZVIZ LB1 Outgoing

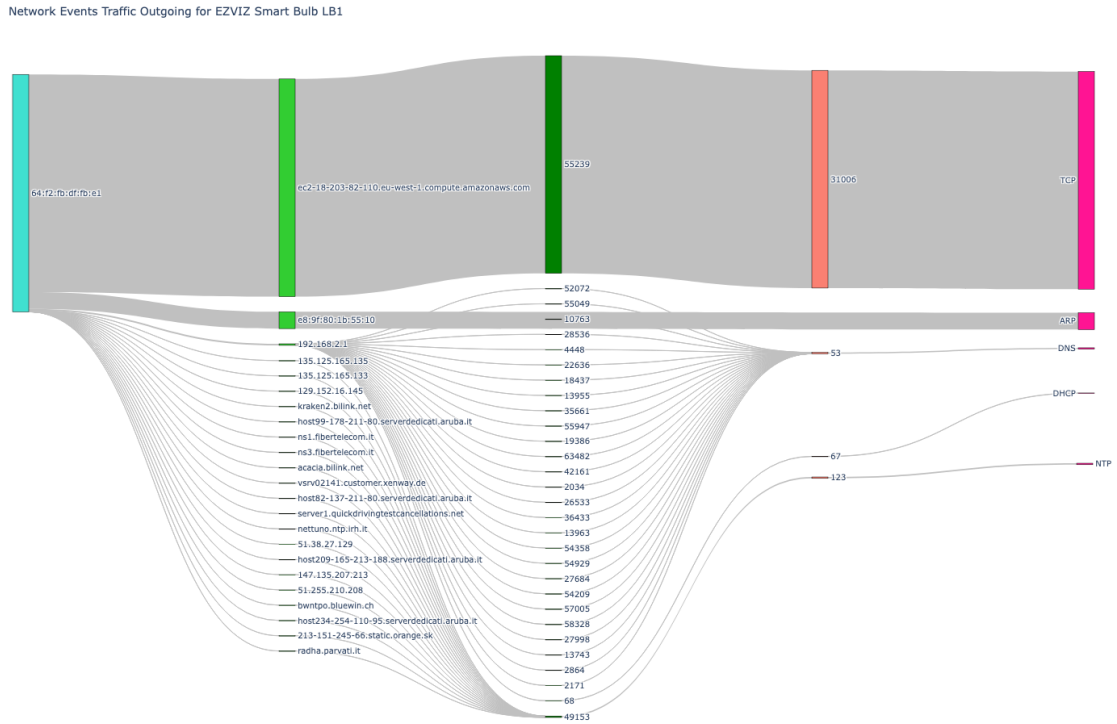

Figure 5.15: Sankey diagram EVENTs Traffic EZVIZ LB1 Outgoing

It could be easily seen that spontaneous traffic exists and takes a not negligible amount of packets. Section 5.2.5 analyzes in detail the ratio between EVENTS and spontaneous traffic over the whole dataset.

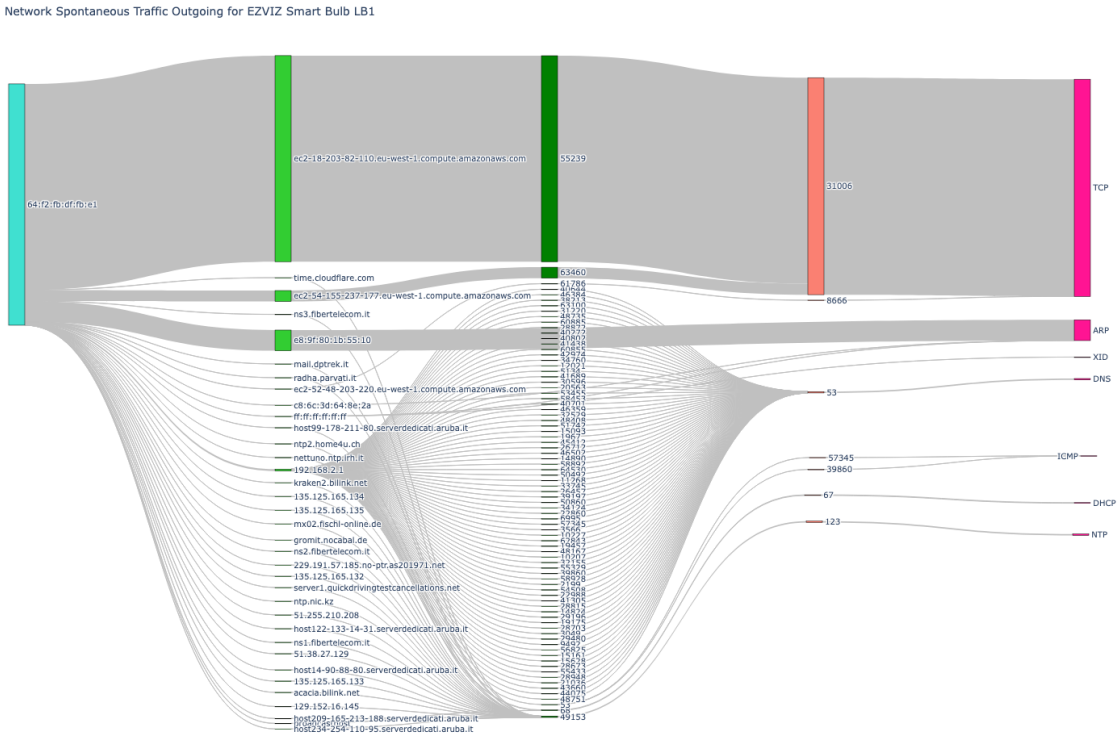

Figure 5.16: Sankey diagram Spontaneous Traffic EZVIZ LB1 Outgoing

5.2.5. Results

For the Sankey diagram were analyzed 72836 packets (10.69 MB), of which 46824 (6.57 MB, 64.3%) are labeled as spontaneous, and 26012 (4.12 MB, 35.7%) as EVENTs.
As quantitative data let's point out the total number of packets generated by each Smart home device.

EZVIZ LB1 Smart bulb

LB1 has generated 33808 network packets, of which 25509 (75.45%) are spontaneous and 8299 (24.55%) are EVENTs-related.

Tapo L530E Smart bulb

L530E has generated 15610 network packets, of which 7330 (46.96%) are spontaneous and 8280 (53.04%) are EVENTs-related.

Tapo P100 Smart plug

P100 has generated 23418 network packets, of which 13985 (59.72%) are spontaneous and 9433 (40.28%) are EVENTs-related.

5.3. Final Observations

EZVIZ smart devices generate a lot of network traffic, both for incoming/outgoing traffic and spontaneous/EVENTs traffic, which is not strictly related to the use of the device. Tapo smart devices generate the strictly necessary traffic, in particular, the L530E smart bulb is the perfect example, while P100 has a little bit of non-essential traffic in the spontaneous packets.

We have demonstrated how it is possible to collect a network traffic dataset automatically by exploiting a mobile app testing framework. We can now answer our questions. The smart devices produce spontaneous (unsolicited) traffic, representing a mean value of 60% of all the network traffic produced by the devices. We have found a linear pattern with which we can discern spontaneous from network traffic.

6 | Conclusions and future developments

This chapter serves as a conclusive section of the work, summarizing the project goal and the results obtained, and presenting future research directions.

6.1. Summary

In this work, we designed, implemented, and presented a new tool, namely AutomIoT, able to automatically generate Android app *EVENTs* to control IoT devices. Such *EVENTs* are state transitions from active interactions between a smart device and its companion app. We have tested the tool by automatically generating *EVENTs* while collecting the network traffic obtaining a large dataset with network data and associated event labels. The Automation was performed using UI Automator, a black-box style mobile testing framework offered by the Android toolchain. The tool we presented can be the baseline for future network dataset collection. Our proposal could improve, facilitate, and make network data generation and collection faster to construct datasets for future IoT Forensics research. In this thesis, we have performed a first approach to IoT Forensics Analysis over the dataset gathered trying to understand whether there exists a kind of spontaneous traffic aside from the traffic generated by *EVENTs* We measure the spontaneous traffic that represents about 60% of the total traffic exchanged by the considered smart devices. We plotted the results using Time Series and Sankey Diagram to show the difference between the two traffic flows (with and without events). The former data visualization technique has highlighted the network packets associated with *EVENTs* and spontaneous traffic, while the latter has shown the characteristics and components inside the traffic being analyzed.

6.2. Future Developments

We can extract a few considerations and open points from our work. First of all, we have exploited UI Automator as a reference framework. We took this decision starting from the Monkey framework used in the related work [46], [47] from which we began our research. UI Automator together with Espresso, are the main reference for Android Testers, with our choice that goes in favor of the former since we needed black-box style kind tests. As shown in Section 2, tons of mobile app testing tools could be used in our scenario. It could be the topic following the thesis trying to compare those testing tools with the same objective in our work and verify what best fits the tackled problem. Other possible future developments starting from this work can be:

- Performing a benchmark between the exceptions handling pattern used by the two aforementioned research works exploiting screenshot matching and our try-catch software code alternative.
- For our Kotlin-based UI Instrumented Test class, it could be possible to create an abstract general class with all the possible interactions that a smart device could be subject to. From this class, it would be possible to instantiate any possible object to create an instrumented test.
- The first time that the EZVIZ mobile app was opened before starting a UI Automator Test, suffered from several crashes before opening correctly and starting the test. Regarding these crashes, we want to highlight that they happened when we were already logged in, otherwise, the app functioned as expected. We did not investigate the details of this issue, but we could guess that they could be related to the emulator environment settings, may be caused by some bugs present in the SDK being used, or from the DNS traffic generated. We tried to use the app inside a real device and those problems disappeared, so it is certainly related to the emulator. We tried also to increase the RAM used by the emulator. The crash ratio decreased, but they were still present. It could be interesting to investigate this issue, understand the causes, and do a comparison with a real device.
- Performing a benchmark over the performance achieved between using an Emulator, as we did, and a real device.
- Expanding our tool to handle more smart devices (e.g. cameras, or smart devices of other brands), to increase their coverage with our instrument.
- Separate the test configuration from the test execution. We mean that it could be

possible to develop a tool able to execute the test following a series of configuration steps without the need to develop any kind of code, as we did. An EVENT generation could be achieved by: first developing an engine that is in charge of writing and executing the code on our behalf and, lastly defining a configuration file, as YAML, XML, or text file in which it could be described as the kind of test that we would like to perform and make automatic. We could see it as a way to facilitate the life of a test developer, reducing the programming skills required. For this consideration, I took inspiration from Ansible [5], [72]. A software that automates the management of remote systems and controls their desired state. Its engine is written in Python.

- If we would like to take to extremes for the previous concept, it could be possible to train a Machine Learning algorithm that gives a test description of the test to run, takes care of or creates the Instrumented Test, and finally executes it.
- Finally, looking up the network dataset, could be conducted other kinds of IoT Forensics Analysis, for example over the traffic to find some vulnerabilities, or analyze the traffic more in-depth to find other evidence out of the scope of our work, or the dataset could be used as a baseline to apply Machine Learning techniques trying to find some correlations. Those correlations could drive the possibility of understanding if a human being is inside a room in a given timestamp for instance, or to understand the type and/or model of a device only looking at the traffic.

Bibliography

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and G. Imparato. A toolset for gui testing of android applications. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 650–653, 2012. doi: 10.1109/ICSM.2012.6405345.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261, 2012. doi: 10.1145/2351676.2351717.

[3] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015. doi: 10.1109/MS.2014.55.

[4] A. S. Ami, M. M. Hasan, M. R. Rahman, and K. Sakib. Mobicomonkey: Context testing of android apps. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '18, page 76–79, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357128. doi: 10.1145/3197231.3197234. URL `https://doi.org/10.1145/3197231.3197234`.

[5] Ansible Project Contributors. Ansible Docs, 2023. URL `https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html`.

[6] APKMirror. APKMirror homepage, 2022. URL `https://www.apkmirror.com/`.

[7] Apkpure. Apkpure playstore, 2022. URL `https://apkpure.com/`.

[8] AQuA. AQuA, 2023. URL `https://www.appqualityalliance.org/aqua-and-deliverables#criteria`.

[9] arXiv. arXiv, 2023. URL `https://arxiv.org/`.

[10] Association for Computing Machinery. Copyright © 2023 ACM, Inc. ACM Digital Library, 2023. URL `https://dl.acm.org/`.

[11] F. Behrang and A. Orso. Seven reasons why: An in-depth study of the limitations of random test input generation for android. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 1066–1077, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450367684. doi: 10.1145/3324884.3416567. URL https://doi.org/10.1145/3324884.3416567.

[12] D. Bonaventura, S. Esposito, and G. Bella. Smart bulbs can be hacked to hack into your household. In *Proceedings of the 20th International Conference on Security and Cryptography*. SCITEPRESS - Science and Technology Publications, 2023. doi: 10.5220/0012092900003555. URL https://doi.org/10.5220%2F0012092900003555.

[13] K. by JetBrains and Google. Kotlin homepage, 2022. URL {https://kotlinlang.org/},{https://kotlinfoundation.org/}.

[14] D. Campos and T. OConnor. Towards labeling on-demand iot traffic. In *Proceedings of the 14th Cyber Security Experimentation and Test Workshop*, CSET '21, page 49–57, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390651. doi: 10.1145/3474718.3474727. URL https://doi.org/10.1145/3474718.3474727.

[15] T. Coelho, B. Lima, and J. a. P. Faria. Mt4a: A no-programming test automation framework for android applications. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, A-TEST 2016, page 59–65, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450344012. doi: 10.1145/2994291.2994300. URL https://doi.org/10.1145/2994291.2994300.

[16] H.-P. Company. *HP Elitebook 830 G5 i5 8250U 1.6GHz CPU, 8 GB RAM, 512 GB PCIe Gen 3x4 NVMe TLC SSD specifications*, 2022. URL https://support.hp.com/us-en/document/c05898590.

[17] S. F. Conservancy. Qemu docs, 2022. URL {https://www.qemu.org/},{https://wiki.qemu.org/Main_Page},{https://www.qemu.org/docs/master/index.html}.

[18] T.-L. Corporation. Tapo Android mobile app playstore, 2022. URL https://play.google.com/store/apps/details?id=com.tplink.iot.

[19] T.-L. Corporation. *Tapo L530E Specifications*, 2022. URL https://www.tp-link.com/it/home-networking/smart-bulb/tapo-l530e/#specifications,

https://www.tapo.com/en/product/smart-light-bulb/tapo-l530e/#tapo-product-spec.

[20] T.-L. Corporation. *Tapo P100 Specifications*, 2022. URL https://www.tp-link.com/it/home-networking/smart-plug/tapo-p100/#specifications,https://www.tapo.com/en/product/smart-plug/tapo-p100,https://www.tapo.com/en/product/smart-plug/tapo-p100/#tapo-product-spec.

[21] Elsevier. Science Direct, 2023. URL https://www.sciencedirect.com/.

[22] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso. Barista: A technique for recording, encoding, and running platform independent android tests. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 149–160, 2017. doi: 10.1109/ICST.2017.21.

[23] I. Glossary. Functional Testing, 2023. URL https://glossary.istqb.org/en_US/term/functional-testing-1-3.

[24] I. Glossary. Non-Functional Testing, 2023. URL https://glossary.istqb.org/en_US/term/non-functional-testing-1-3.

[25] I. Glossary. Performance Testing, 2023. URL https://glossary.istqb.org/en_US/term/performance-testing-2-2.

[26] I. Glossary. Security Testing, 2023. URL https://glossary.istqb.org/en_US/term/security-testing.

[27] I. Glossary. Usability Testing, 2023. URL https://glossary.istqb.org/en_US/term/usability-testing-4-2.

[28] G. Hu, L. Zhu, and J. Yang. Appflow: Using machine learning to synthesize robust, reusable ui tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 269–282, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236055. URL https://doi.org/10.1145/3236024.3236055.

[29] ImageMagick Studio LLC. ImageMagick, 2023. URL https://imagemagick.org/.

[30] A. Inc. *MacBook Pro (Retina, 13-inch, Early 2015) Technical Specifications*, 2021. URL https://support.apple.com/kb/SP715?locale=en_US.

[31] E. Inc. Ezviz Android mobile app playstore, 2022. URL https://play.google.com/store/apps/details?id=com.ezviz.

[32] E. Inc. *Ezviz LB1 Color Specifications*, 2022. URL https://www.ezviz.com/it/product/LB1-Color/28496.

[33] E. Inc. *Ezviz T31 Specifications*, 2022. URL https://www.ezviz.com/it/product/T31/16490.

[34] Institute of Electrical and Electronics Engineers (IEEE). IEEE Explore, 2023. URL https://ieeexplore.ieee.org/Xplore/home.jsp.

[35] IoT Analytics. IoT 2022 in review: the 10 most relevant iot developments of the year, 2023. URL https://iot-analytics.com/iot-2022-in-review/.

[36] IoT Analytics. Global IoT market size to grow 19despite economic downturn, 2023. URL https://iot-analytics.com/iot-market-size/.

[37] IoT Analytics. State of IoT 2023: number of connected iot devices growing 1616.7 billion globally, 2023. URL https://iot-analytics.com/number-connected-iot-devices/.

[38] Jupyter. *Jupyter Notebook Docs*, 2022. URL https://jupyter-notebook.readthedocs.io/en/latest/.

[39] A. Kapoor and S. Raza Qureshi. Forensic analysis of digital evidence extracted from amazon echo. In *2020 IEEE International Conference on Advent Trends in Multidisciplinary Research and Innovation (ICATMRI)*, pages 1–7, 2020. doi: 10.1109/ICATMRI51801.2020.9398391.

[40] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68 (1):45–66, 2019. doi: 10.1109/TR.2018.2865733.

[41] Y. Li, Z. Yang, Y. Guo, and X. Chen. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1070–1073, 2019. doi: 10.1109/ASE.2019.00104.

[42] I. Linksys Holdings. *Linksys datasheet*, 2016. URL https://downloads.linksys.com/downloads/datasheet/WRT3200ACM_WiFiRouter_EN.pdf.

[43] I. Linksys Holdings. *Linksys user guide*, 2016. URL https://downloads.linksys.com/downloads/userguide/WRT3200ACM__UG_International.pdf.

[44] C. H. Liu, C. Y. Lu, S. J. Cheng, K. Y. Chang, Y. C. Hsiao, and W. M. Chu. Capture-replay testing for android applications. In *2014 International Symposium*

on Computer, Consumer and Control, pages 1129–1132, 2014. doi: 10.1109/IS3C.2014.293.

[45] Y. Ma, Y. Huang, Z. Hu, X. Xiao, and X. Liu. Paladin: Automated generation of reproducible test cases for android apps. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, HotMobile '19, page 99–104, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362733. doi: 10.1145/3301293.3302363. URL `https://doi.org/10.1145/3301293.3302363`.

[46] A. M. Mandalari, R. Kolcun, H. Haddadi, D. J. Dubois, and D. Choffnes. Towards automatic identification and blocking of non-critical iot traffic destinations, 2020.

[47] A. M. Mandalari, D. J. Dubois, R. Kolcun, M. T. Paracha, H. Haddadi, and D. Choffnes. Blocking without breaking: Identification and mitigation of non-essential iot traffic, 2021.

[48] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk. Crashscope: A practical tool for automated testing of android applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 15–18, 2017. doi: 10.1109/ICSE-C.2017.16.

[49] I. C. Morgado and A. C. R. Paiva. The impact tool for android testing. *Proc. ACM Hum.-Comput. Interact.*, 3(EICS), jun 2019. doi: 10.1145/3300963. URL `https://doi.org/10.1145/3300963`.

[50] S. Negara, N. Esfahani, and R. Buse. Practical android test recording with espresso test recorder. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 193–202, 2019. doi: 10.1109/ICSE-SEIP.2019.00029.

[51] Numpy. Numpy homepage, 2022. URL `https://numpy.org/`.

[52] Open Handset Alliance and commercially and sponsored by Google. *Android Debug Bridge (ADB) Docs*, 2022. URL `https://developer.android.com/studio/command-line/adb`.

[53] Open Handset Alliance and commercially and sponsored by Google. *Android Emulator Docs*, 2022. URL `{https://source.android.com/docs/devices/automotive/start/avd},{https://source.android.com/docs/setup/create/avd},{https://developer.android.com/studio/run/emulator},{https://developer.android.com/studio/run/emulator-commandline}`.

[54] Open Handset Alliance and commercially and sponsored by Google. *Android Studio Docs*, 2022. URL https://developer.android.com/studio/intro.

[55] Open Handset Alliance and commercially and sponsored by Google. *Create and manage virtual devices Docs*, 2023. URL https://developer.android.com/studio/run/managing-avds.

[56] Open Handset Alliance and commercially and sponsored by Google. *espresso*, 2023. URL https://developer.android.com/training/testing/espresso.

[57] Open Handset Alliance and commercially and sponsored by Google. *Fundamentals of testing Android apps*, 2023. URL https://developer.android.com/training/testing/fundamentals.

[58] Open Handset Alliance and commercially and sponsored by Google. *UI/Application Exerciser Monkeyr*, 2023. URL https://developer.android.com/studio/test/other-testing-tools/monkey.

[59] Open Handset Alliance and commercially and sponsored by Google. *monkeyrunner*, 2023. URL https://developer.android.com/studio/test/monkeyrunner.

[60] Open Handset Alliance and commercially and sponsored by Google. *UI Automator Docs*, 2023. URL https://developer.android.com/training/testing/other-components/ui-automator.

[61] OpenWrt. OpenWrt project, 2022. URL https://openwrt.org/.

[62] OpenWrt. OpenWrt Tcpdump docs, 2022. URL https://openwrt.org/docs/guide-user/firewall/misc/tcpdump_wireshark.

[63] F. Palmese. *Feature Sniffer Docs*, 2022. URL https://github.com/fpalmese/feature-sniffer.

[64] I. H. b. O. Pandas via NumFOCUS. Pandas homepage, 2022. URL {https://pandas.pydata.org/},{https://numfocus.org/},{https://www.ovhcloud.com/it/}.

[65] Plotly. Plotly homepage, 2022. URL https://plotly.com/.

[66] A. C. Re. 01 Time Series Analysis Smart Device Network Traffic (SDNT), 2023. URL https://github.com/rhacrsse/AutomIoT/blob/main/linksysAnalysis/01.timeseriesanalysisSDNT.ipynb.

[67] A. C. Re. 02 Sankey Diagram Smart Device Network Traffic (SDNT), 2023. URL

https://github.com/rhacrsse/AutomIoT/blob/main/linksysAnalysis/02.SankeydiagramSDNTsingle.ipynb.

[68] A. C. Re. 03 Sankey Diagram Smart Device Spontaneous Network Traffic (SDSNT), 2023. URL https://github.com/rhacrsse/AutomIoT/blob/main/linksysAnalysis/03.SankeydiagramSDSNTsingle.ipynb.

[69] A. C. Re. 04 Sankey Diagram Smart Device EVENTS Network Traffic (SDENT), 2023. URL https://github.com/rhacrsse/InOT-Forensics-Android-UI-Automation-Testing/blob/main/linksysAnalysis/04.SankeydiagramSDENTsingle.ipynb.

[70] A. C. Re. Mozart Orchestrator, 2023. URL https://github.com/rhacrsse/UI-Automator-Functional-Tests.

[71] A. C. Re. Walkthrough, 2023. URL https://github.com/rhacrsse/AutomIoT/blob/main/WIP/guide.md.

[72] Red Hat, Inc. Ansible, 2023. URL https://www.ansible.com/.

[73] S. J. Saidi, A. M. Mandalari, R. Kolcun, H. Haddadi, D. J. Dubois, D. Choffnes, G. Smaragdakis, and A. Feldmann. A haystack full of needles: Scalable detection of iot devices in the wild. In *Proceedings of the ACM Internet Measurement Conference*, IMC '20, page 87–100, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381383. doi: 10.1145/3419394.3423650. URL https://doi.org/10.1145/3419394.3423650.

[74] S. J. Saidi, A. M. Mandalari, H. Haddadi, D. J. Dubois, D. Choffnes, G. Smaragdakis, and A. Feldmann. Detecting consumer iot devices through the lens of an isp. In *Proceedings of the Applied Networking Research Workshop*, ANRW '21, page 36–38, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386180. doi: 10.1145/3472305.3472885. URL https://doi.org/10.1145/3472305.3472885.

[75] I. Santos, J. C. C. Filho, and S. R. S. Souza. A survey on the practices of mobile application testing. In *2020 XLVI Latin American Computing Conference (CLEI)*, pages 232–241, 2020. doi: 10.1109/CLEI52000.2020.00034.

[76] I. Sauce Labs. Getting Started with Mobile App Testing, 2022. URL https://saucelabs.com/resources/blog/guide-to-mobile-app-testing.

[77] M. Stoyanova, Y. Nikoloudakis, S. Panagiotakis, E. Pallis, and E. K. Markakis. A survey on the internet of things (iot) forensics: Challenges, approaches, and open

issues. *IEEE Communications Surveys and Tutorials*, 22(2):1191–1221, 2020. doi: 10.1109/COMST.2019.2962586.

[78] M. D. Team. Matplotlib homepage, 2022. URL https://matplotlib.org/.

[79] TestingXperts. A Detailed Mobile Application Testing Guide, 2023. URL https://www.testingxperts.com/blog/mobile-application-testing.

[80] O. Thompson, A. M. Mandalari, and H. Haddadi. Rapid iot device identification at the edge. In *Proceedings of the 2nd ACM International Workshop on Distributed Machine Learning*, DistributedML '21, page 22–28, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450391344. doi: 10.1145/3488659.3493777. URL https://doi.org/10.1145/3488659.3493777.

[81] UNSW Sydney. Amazon Echo Sankey Diagram, 2023. URL https://iotanalytics.unsw.edu.au/profiles.html.

[82] I. K. Villanes, E. A. Bezerra Costa, and A. C. Dias-Neto. Automated mobile testing as a service (am-taas). In *2015 IEEE World Congress on Services*, pages 79–86, 2015. doi: 10.1109/SERVICES.2015.20.

[83] S. Virciglio, A. Redondi, and F. Palmese. Performance evaluation of online traffic feature extraction for iot forensics. Master's thesis, ING - Scuola di Ingegneria Industriale e dell'Informazione, 12 2021. URL http://hdl.handle.net/10589/182790.

[84] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie. An empirical study of android test generation tools in industrial cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 738–748, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359375. doi: 10.1145/3238147.3240465. URL https://doi.org/10.1145/3238147.3240465.

[85] Wireshark. *Tshark Docs*, 2022. URL https://www.wireshark.org/docs/man-pages/tshark.html.

[86] Wireshark. Wireshark homepage, 2022. URL https://www.wireshark.org/.