POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

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

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Angelo Claudio Re

Advisor: Prof. Alessandro Cesare Enrico Redondi

Co-advisor: Dott. Fabio Palmese

Academic year: 2022-2023

Abstract

IoT forensics is a new branch of digital forensics, gaining attention due to the rapid growth of IoT devices. Its goal is to extract information from such devices, to be later used for investigations. However, there are many challenges in the specific field of IoT network forensics. One of these is the lack of labeled datasets, from which it could be extracted reliable sources of evidence. Such datasets should contain both the network traffic and the device activity in the form of (*"EVENTs"*). Building such datasets requires much time and human effort, bringing our focus to this work. This work presents AutomIoT, a framework able to automate the interaction between a smart device and its companion app to produce a labeled network dataset using UI Automator framework. The tool generates events while the network traffic is being acquired, and all the operations performed are associated with an activity log, allowing us to label the obtained dataset. Finally, we analyze the existence of "spontaneous" traffic, discerning it from those related to the devices *"EVENTs"*. We refer to different data visualization techniques giving a quantitative measure of the spontaneous traffic, that represents about 60% of the whole dataset.

1. Introduction

IoT forensics is a new branch of digital forensics that aims to extract information from devices in the field of the Internet of Things. It can be very challenging due to the nature of IoT devices and the heterogeneous environment in which they operate. Depending on where the information is gathered, IoT forensics deals with different possible sources of evidence: our work focuses on network traffic extracted from such devices. In the specific field of network forensics, gathering data related to IoT devices has a crucial role in analyzing network traffic to discover possible vulnerabilities and also to spot some evidence that could be relevant in investigations. Our research focuses on IoT smart home devices and their network traffic. When dealing with network traffic for forensics purposes, the major limitation highlighted in the literature is the lack of labeled datasets that can be used for extracting evidence using ML/DL techniques. To apply ML techniques, such datasets should contain network traffic and a label describing the device activity, if any. Our work poses the goal of solving this gap, presenting a tool able to automate the *EVENTs* generation to produce a labeled network dataset, using the UI Automator frame-

work to trigger the devices in the smart home without involving human effort. The operations performed while the traffic is collected are then associated with an activity log generated by the proposed framework, allowing us to label the obtained network traffic traces and build the final dataset. After describing the tool, we built our dataset containing traffic traces associated with activity logs from four different devices and analyzed it to inspect the presence of spontaneous traffic, which is made up of network data generated when a smart device is in its state and is not triggered by the user. To conclude the work, we use different data visualization tools to measure the spontaneous traffic phenomenon by correlating the activity log and the network traffic dataset. The rest of the work is organized as follows: Section 2 comments IoT Forensics and Mobile App Testing Automation related work, Section 3 describes the architecture of our system and the implementation of the tool that we have developed, Section 4 presents the results obtained, Section 5 are the conclusions.

2. Related Work

Our work is connected to the IoT Network Forensics subdomain and Mobile App Testing fields. We selected two works concerning both topics as a starting point for our research. These papers involve the discovery and blocking of non-essential IoT traffic destinations without breaking the normal functioning of the mobile app involved[2], [1]. The authors propose a solution to manage network traffic generated by smart devices and their companion apps to increase privacy and security. They classified the network traffic as either critical or not critical. The critical label is associated with the traffic essential for the smart devices to operate their main functions correctly. They have also implemented two complementary methods able to generate, identify (*IoTrigger*) and block (*IoTrimmer*) non-critical traffic. To generate network traffic from smart devices, they exploited the Monkey [1] framework offered by Android Studio. We consider this work the baseline for the tool proposed in this thesis since our primary goal is to automate the *"EVENTs"* generation to automatically generate activity from

[1] https://developer.android.com/studio/test/other-testing-tools/monkey

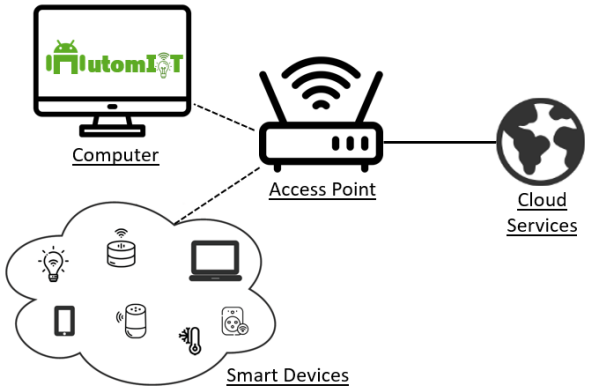

Figure 1: Overall Architecture

IoT devices and collect a labeled dataset. The authors also provided a way to handle possible failures in the interaction between the two components, using screenshot matching. In particular, they compare the screenshots taken before firing the sequence of events using Monkey, with the screenshots registered after the events have been executed. If the matching is correct, it means the device has been triggered successfully. As we will see later in this work, we avoided implementing the same idea but we handled this check directly in our code.

3. Project Overview

This section presents the main components of the AutomIoT framework: system architecture, UI Automator test development, and the data gathering process.

3.1. Architecture

Figure 1 sketches the architecture of the proposed system and how the different entities communicate in the network. Two macro elements constitute the architecture: the local network where all the devices are located, as well as the computer running the AutomIoT software for Android emulation, and the remote servers, which usually take care of all the functionalities executed by the IoT devices, checking their operations and logging the activities, optionally storing relevant data. The head of the framework is the computer, which is used as a mobile device emulator for the UI Test automation.

3.2. Implementation

As discussed in previous sections, to collect a labeled dataset containing network traffic with

the activity log, we need a tool to automate the event generation using the Android applications to control the smart devices. The generated events induce state changes in the paired smart device, usually causing the generation of specific network traffic patterns for communication with the remote servers that should check/execute/log the operations. A few works in the literature have shown that it is possible to automate Android interactions with Monkey, an Android tool usually involved in mobile testing to generate pseudo-random events to see the robustness and check the functionalities of the mobile application tested. Monkey is a command-line-based tool that exploits the ADB [3] command line program to launch the tests. In our scenario, we choose and adopt the UI Automator framework that allows us to conduct cross-app functional black-box style UI Instrumented Unit tests, but with complete control over the test execution to assign the label correctly to the network dataset. It also belongs to the Android toolchain as Monkey, and along with Espresso [2], it is the primary tool Android developers use for usual testing operations before the app release. The UI Automator framework can be programmed using Kotlin or Java: in our case, we use the Kotlin programming language to implement the AutomIoT system functions. To develop a UI Instrumented Unit Test, it is necessary to initialize a UiDevice object and to collect a UISelector with a GUI inspection tool. The UiDevice object provides access to state information about the device: it represents the core component used to interact with a UIElement exploiting a UISelector. By using a proper method (findObject) and passing as input the selector, the UIDevice can interact with a UIElement and emulate gestures performed on the screen of the emulated device. Figure 2 reports some examples of the gestures that can be produced. Alternatively, it is possible to use the click method to generate the tap gesture, passing the pixel position of the UIElement that we want to trigger as input, referring to the (x,y) coordinates. The framework can use UISelectors to identify, detect, and select the UIElements that are parts of a mobile application that can be interacted with. The selectors are identifiers the developer specifies during the app development

[2]https://developer.android.com/training/testing/espresso

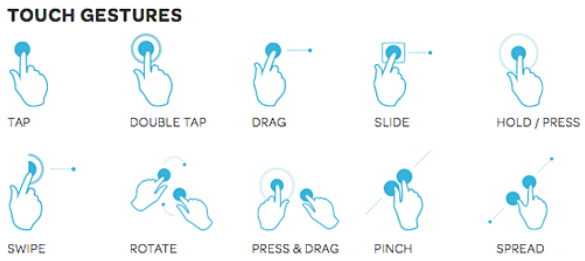

Figure 2: Example of Gesture icons

process (e.g., resource ID, text labels), or the UIElement pixel positions. They can univocally identify UIElements in the application: we manually discover them by exploiting UIAutomatorViewer, which is an Android GUI inspection tool that provides a convenient visual interface to inspect the layout hierarchy (an XML file), named Views, and shows the properties of UI components that are visible on the foreground of the device. This way, we can create a UI selector matching a specific visible property. Figure 3 depicts an example of UIAutomatorViewer GUI: in this case, we show the part of the Tapo mobile application and the area portion of a UIElement with the corresponding coordinates that can be used for generating the required gestures.

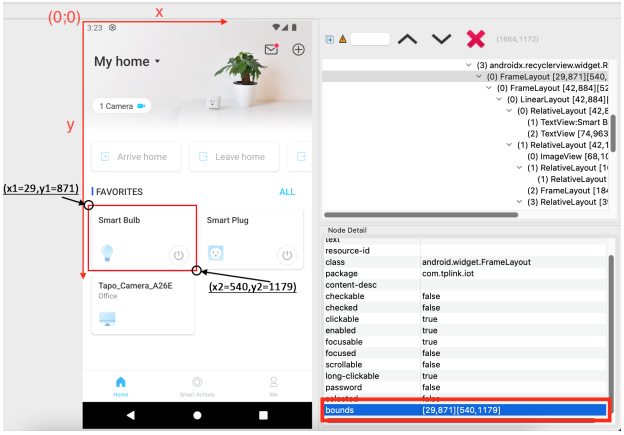

Figure 3: UIAutomatorViewer Example

Considering the previous basic blocks, we create one Kotlin class for each unique device to generate the events. This choice was driven by two factors: (i) the app can have different behaviors during the navigation of the various Views to reach the one with the required UI Element to trigger, and (ii) the action needed to be executed to trigger the UI Element usually differ from device to device. Given this, there is no

general pattern to generate the events for all the devices, so we create one different class per device to implement all the events that a single device can support. Each class is equipped with some standard variables, attributes, and methods, used to control the device: 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 device's state must be updated depending on the action performed. The methods, instead, allow us to perform actions, such as view navigation, or event triggers. After implementing all the methods required to trigger all the functionalities for each device, two essential elements of the development should be given particular care: the activity logs and how to handle possible exceptions that could arise during the execution of the tests. For what concerns the activity log, after executing a method, a shared file is used to store information on all the activities generated by smart devices. After the test execution, we exploit the ADB tool to extract the log from the emulator after the data-gathering phase. Exception handling is essential to decrease the source of test crashes and avoid misalignment between the state of the smart device and the companion app, which usually are related to backend problems or app feedback popups. For this reason, we introduce several countermeasures as follows:

- We introduced a time delay set empirically to 5 seconds that gives time to the Android View to be loaded, before generating the first event of the test.
- We added a time delay set empirically to 2 seconds between the different generated events. In this way, the tests await a certain amount of time for the smart device to apply the change of state correctly before proceeding to the next event, giving time to the cloud server to check and reply to the state change.
- We use "try-catch" statements all over the software parts that trigger an event. If something goes wrong, the test keeps working properly for next events, and a NOP operation is added to the activity log, notifying the error, instead of the correct event log.
- A proper function is introduced to recognize when a popup View appears over the device's screen and to close it accordingly, allowing the test to continue as expected.

4. Experimental Results

This section presents the testbed we referred to for evaluating our framework, the primary data analysis techniques used for the work, and the experimental results obtained from the different smart devices considered. We set up a local network with four different Smart Home devices produced by two different brands: Tapo L530E Smart Bulb, Tapo P100 Smart Plug, Ezviz LB1 Smart Bulb, and Ezviz T31 Smart Plug. The firmware is proprietary software for all four devices, and each device can be controlled from the corresponding proprietary application: Ezviz and Tapo. All the devices are powered on, and connected to the Access Point at the beginning of the tests. To capture the network traffic while generating the events, we used the LinkSys WRT3200ACM Access Point, which supports the OpenWrt firmware [4] and can run tcpdump [3] to sniff the network traffic. We refer to an open-source network feature extraction framework to capture the interested features directly [5] for the selected devices, filtering the traffic by MAC address, and computing statistics on a time base with windows of X seconds. The tests are executed for 48 consecutive hours approximately, generating a dataset containing more than 3000 events. The network traffic is stored in a different folder for each device, producing an outcome of one PCAP and one CSV per device. During the event generation, the NOP operations were 116 over 3000 events, meaning that less than 4% of the activities incurred errors. For the data analysis, we correlated the network dataset with the activity log to assign a label to the data. We performed a twofold kind of analysis: we created Time Series plots to understand how the network traffic changes concerning the events, and we created Sankey Diagrams to discover how many packets were associated with events and spontaneous traffic what IPs/services or ports were used. The analysis has been conducted using Python libraries, filtering the data by the smart devices'

[3] https://www.tcpdump.org/

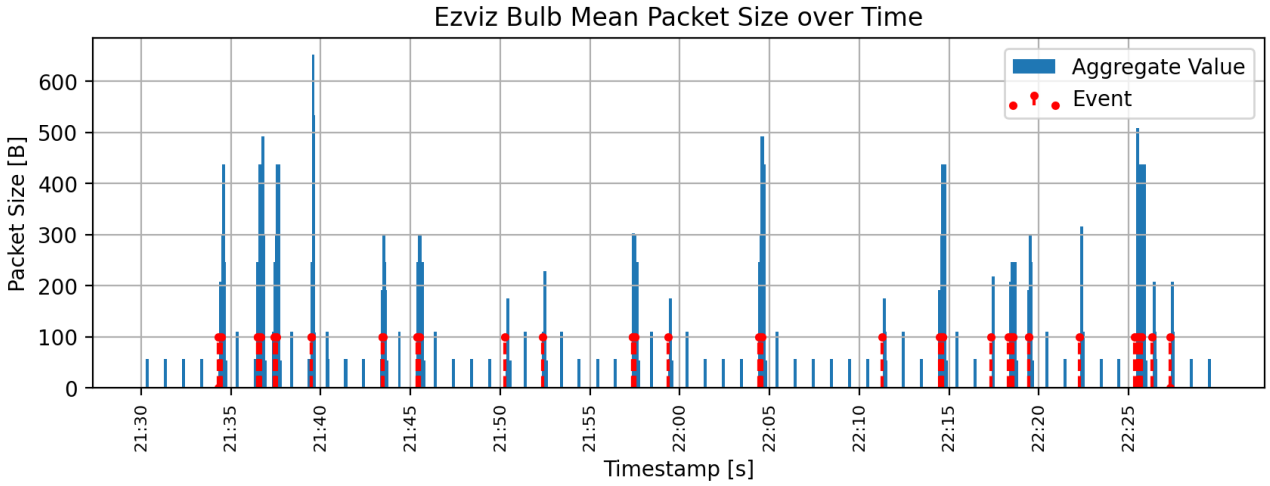

Figure 4: Ezviz LB1 Time Series Example

Mac addresses.

We have aggregated data using different time window values (0.5, 1, 5, and 10 seconds) with the Feature-Sniffer tool. We have compared the four different window values and created the corresponding time series plots. Based on the qualitative appearance of the different plots, we decided to report the results for the case using 2-second time windows. The behavior of the time series for all four devices is very similar. To avoid redundancy and reduce space, we focus only on one of the four devices: the Ezviz Light Bulb. To discern EVENT's Network Traffic (ENT) and Spontaneous Network Traffic (SNT) we use a time window of 2 seconds in the proximity of the event generated. This value is not the aggregation time window, but another window used to discriminate what packets are considered in the events and what are instead associated with spontaneous traffic. In this way, if the event occurred at time x, all packets in the range [x-1,x+1] are considered generated by the event, while the rest are associated with spontaneous traffic. This value was chosen considering the delay in the event generation phase. Figure 4 reports the time series generated using 2-second time windows for the Ezviz Bulb devices, selecting the mean packet size as the aggregate value. The improve the visibility, we selected only 1 hour of traffic to highlight the relation between the aggregate value and the events. The blue bars in the plot represent the aggregated value, while the red lines report the time when an EVENT occurred for the device. As we can see, the red bars usually correspond to a spike in the bar plot. From the plot, we can conclude that a simple way exists to distinguish event-associated traffic from spontaneous traffic. Indeed, a simple threshold-oriented approach can distinguish if the traffic is produced by generating an event, or if the device is spontaneously transmitting packets to the cloud servers. For example, for the Ezviz smart bulb, a threshold of 150 Bytes in the average packet size of 2-second windows can represent a good division between ENT and SNT, meaning that all window values above the threshold can be considered as ENT, and the rest as SNT.

To conclude our analysis, we have analyzed possible differences in Sankey Diagrams from packets sent (outgoing flow) and received (incoming flow) for the smart devices, under events and silent periods. A Sankey diagram graphically represents the network traffic by aggregating the packets by protocol, transport layer port, and destination. Figure 5 reports the Outgoing Sankey Diagram of the Ezviz Smart Bulb device under silent periods.

The Sankey of Ezviz brand devices SNT produced a lot of network traffic compared to Tapo. This could be why the emulated device suffered from random crashes during the Ezviz app execution. Using an actual device could solve this problem, which was partially countermeasured by increasing the RAM used by the emulator. Outgoing and Incoming traffic have identical behaviors. Different devices from the same brand do not show a significant change and result in

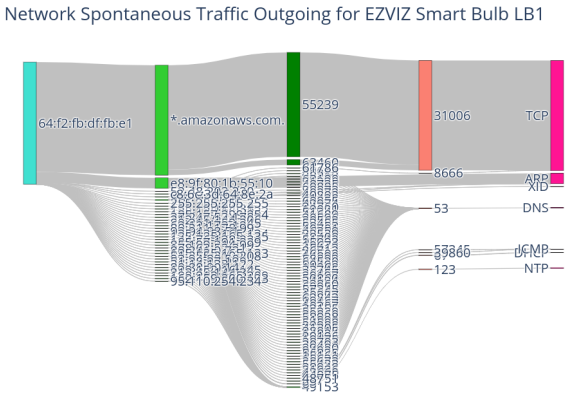

Figure 5: Ezviz LB1 Outgoing Sankey Diagram SNT Example

very similar diagrams. Instead, for what concerns the same device but from different brands (e.g., the two bulbs), we see that there is a lot of DNS traffic and noise traffic, maybe related to the integration of external services, or third-party analytics services as spotted for instance in the papers presented in section 2. The main aspect of spotting here is using AWS web services to host the backend of the smartphone applications for both Tapo and Ezviz. There is no direct interaction between the device and the controlling application, but everything passes through the Internet, given that no local destination is contacted. We can point out the total packet size generated by Ezviz LB1 Smart Bulb as quantitative data. The results compared to other smart devices are very similar. For the Sankey Diagram, we analyzed more than 70k packets, of which 45k are labeled as spontaneous, and 25k are related to EVENTs. To summarize the obtained results, Ezviz smart devices generate a lot of network traffic, which is not strictly related to the use of the device, which may have a significant impact on energy consumption and forensic analysis tasks. Tapo smart devices are simpler logic-wise: they usually generate the strictly necessary traffic, only when events trigger them. We have demonstrated how collecting a network traffic dataset automatically is possible by exploiting a mobile app testing framework. We can now answer our questions. The smart devices produce spontaneous (unsolicited) traffic, representing more than 60% of all the network traffic the device produces. Finally, we have found a simple methodology based on threshold approaches, with which we can distinguish spontaneous from network traffic, detecting when the device is being used only from network traffic

5. Conclusions

In this work, we designed and tested AutomIoT, a new tool that automatically generates *"EVENTs"* that allows the collection of labeled network data for Smart Home devices. After presenting the framework logic and its implementation, we collected our dataset and proposed an analysis of four smart home devices. We study the network traffic from the devices and try to understand if network traffic exists when the devices are not explicitly triggered by the user, trying to understand a relation between the traffic and the triggered events. We measure the spontaneous traffic that represents about 60% of the total traffic exchanged by the considered smart devices. The tool we presented can be the baseline for future works to build new labeled network datasets, for Machine Learning forensics analysis. It is possible to extend the framework to work for other smart devices not considered in this proposal, to be able to work in different IoT networks.

References

[1] Anna Maria Mandalari, Daniel J. Dubois, Roman Kolcun, Muhammad Talha Paracha, Hamed Haddadi, and David Choffnes. Blocking without breaking: Identification and mitigation of non-essential iot traffic, 2021.

[2] Anna Maria Mandalari, Roman Kolcun, Hamed Haddadi, Daniel J. Dubois, and David Choffnes. Towards automatic identification and blocking of non-critical iot traffic destinations, 2020.

[3] Open Handset Alliance and commercially and sponsored by Google. *Android Debug Bridge (ADB) Docs*, 2022.

[4] OpenWrt. OpenWrt project, 2022.

[5] Fabio Palmese, Alessandro EC Redondi, and Matteo Cesana. Feature-sniffer: Enabling iot forensics in openwrt based wi-fi access points. In *2022 IEEE 8th World Forum on Internet of Things (WF-IoT)*, pages 1–6. IEEE, 2022.