UCLA CS211 17F Project Final Report

Universal Analytics for Wi-Fi and Cellular

Group Number: G1

Zongheng Ma
UCLA
zma@cs.ucla.edu

Cyrus Tabatabai-Yazdi
UCLA
ctabatab@ucla.edu

Quanjun Pan
UCLA
aaronpan@ucla.edu

Jason Zheng
UCLA
jwzheng@ucla.edu

ABSTRACT

In this project we developed a new plug-in for MobileInsight, a mobile software that is currently capable of analyzing the status of cellular networks.[4]. Considering the prevalence of Wi-Fi access through mobile devices, this plug-in allows users to perform Wi-Fi network analytics and to gain low-level, detailed information about the Wi-Fi Network. Our goal through this plug-in is providing a better platform for developers and researchers to conduct their evaluations and analysis about Wi-Fi networks. We implemented this plug-in in Python and we also performed analysis on the results we obtained through our plug-in.

1 INTRODUCTION

With the increasing popularity of smart phones and mobile networks, mobile devices are doubling their data traffic every year and there is no sign indicating this trend will end in the near future[2]. In particular, Wi-Fi will only continue to grow as more hotspots pop up and Wi-Fi becomes more accessible in many more places. As shown in `Figure 1`, more than 50% of mobile data are accessed through Wi-Fi network in every country surveyed for both iOS and Android[7]. Based on `Figure 2`, Wi-Fi is also the major data source in the USA[5]. At the same time, a large amount of money has been invested to deploy machine-to-machine network among billions of machines such as automobiles, which will further increase mobile traffic. These Vehicular Ad Hoc Networks will support communication using LTE, Wifi, or 5G and, considering the amount of vehicles on the road today, it is safe to assume mobile data traffic will grow exponentially.

Consequently, traditional cellular networks are facing the challenges brought up by this mobile data explosion. To handle such voluminous mobile data, network providers and architects constructed a hybrid network architecture to migrate data traffic from cellular networks to Wi-Fi access points (APs). However, the transition from a Wi-Fi hotspot to cellular coverage is often very abrupt and even leads to unacceptable disconnection. Given the above scenario, understanding how Wi-Fi networks incorporate with cellular networks behind the scenes in mobile networks is crucial for developers and researchers to optimize the performance of current mobile networks. In addition, some new proposed technologies like MPTCP depend on understanding the interplay of cellular and Wi-Fi[8]. If a dual homed device wants to choose what interface to send out of, it has to analyze both Wi-Fi and cellular interfaces in order to make a good decision. MPTCP is a variant of TCP that can transmit out of multiple interfaces, say LTE and Wi-Fi, simultaneously for increased throughput or choose the best one at a given time. To make the optimal decision, we have to know the status of the respective networks which requires understanding both cellular and Wi-Fi networks at a low-level where the information needed to make optimal decisions is located. Thus, it is imperative to developers to be able to access the low-level information needed to develop new applications and technologies. However, due to the lack of the tools, users and developers have very limited access to the information regarding the status of their mobile networks.

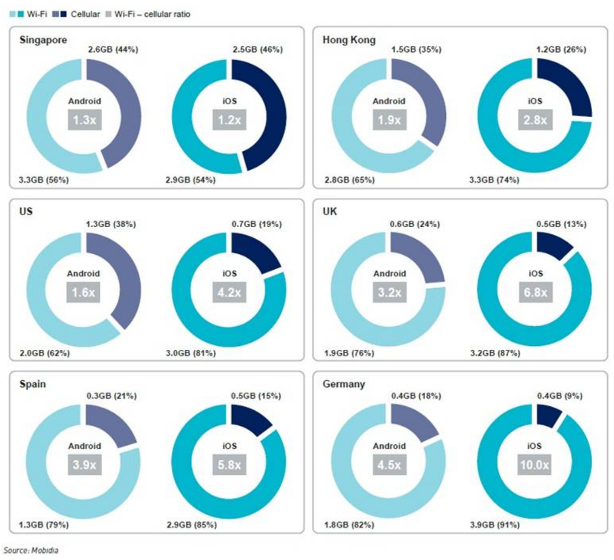
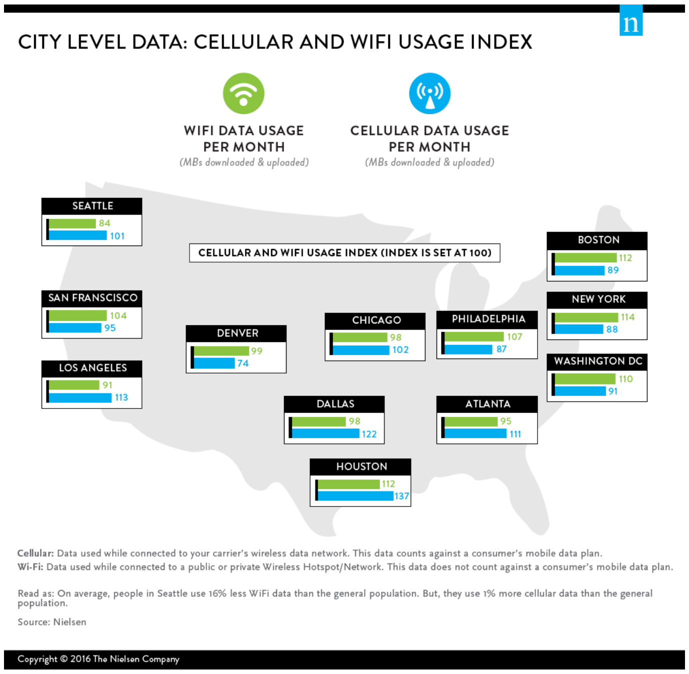


Figure 1: Wi-Fi Data Percentage in major cities of the USA[5]

To resolve this issue, MobileInsight was introduced. It is an application on the Android Operating System that present users with in-depth information about the status of the runtime cellular network, allowing developers to look behind the curtains to see how data is being transferred on cellular networks[4].

Despite the capabilities of MobileInsight on cellular networks, it does not support Wi-Fi networks at this time. Given the important role Wi-Fi networks play in mobile devices, we developed a plug-in to generalize the application to analyze both Wi-Fi information and Cellular information, and create an application capable of handling and understanding the vast majority of wireless traffic for mobile



Cellular: Data used while connected to your carrier's wireless data network. This data counts against a consumer's mobile data plan.
Wi-Fi: Data used while connected to a public or private Wireless Hotspot/Network. This data does not count against a consumer's mobile data plan.

Read as: On average, people in Seattle use 16% less WiFi data than the general population. But, they use 1% more cellular data than the general population.

Source: Nielsen

Copyright © 2016 The Nielsen Company

Figure 2: Wi-Fi Data Percentage by Country[7]

phones and applications. This will make MobileInsight a one-stop-shop for wireless network analytics.

MobileInsight supports third-party plug-ins without modifying the source code of the application. We deployed our new functionalities by implementing a new plug-in written in Python with the Kivy python-for-android framework[9] in order to extract the Wi-Fi information.

In our plug-in, we acquired Wi-Fi information through wpa supplicant, a network supplicant pre-installed on Android devices, and Android built-in APIs. We present our results by directly showing them on the screen in MobileInsight and we also save the results to a log file.

Moreover, we performed analytics on the data we collected on the provided device by running our plug-in. By measuring the delay between Wi-Fi events such as association, associated, connection, and disconnection, we were able to paint a picture on the dynamic nature of Wi-Fi connectivity to be able to help developers infer where latency may be coming from as well as the low-level events that are occuring.

2 PROBLEM DEFINITION AND SCOPE

The problem we are trying to solve is to make it easier for developers and others to analyze runtime network information from Wi-Fi networks. Currently, it is hard to find Wi-Fi network analytics that go beyond signal strength and link speed to more accurately model the dynamics of Wi-Fi networks. Our goal is to provide these fresh analytics and information that can provide low-level information hard to find elsewhere. There is a lot of important information that can only be gleaned by looking at low-level information. However, the existing MobileInsight suite provides for analysis of cellular networks, but with the prevalence of Wi-Fi hotspots and ubiquitous Wi-Fi connectivity, being able to analyze Wi-Fi networks in

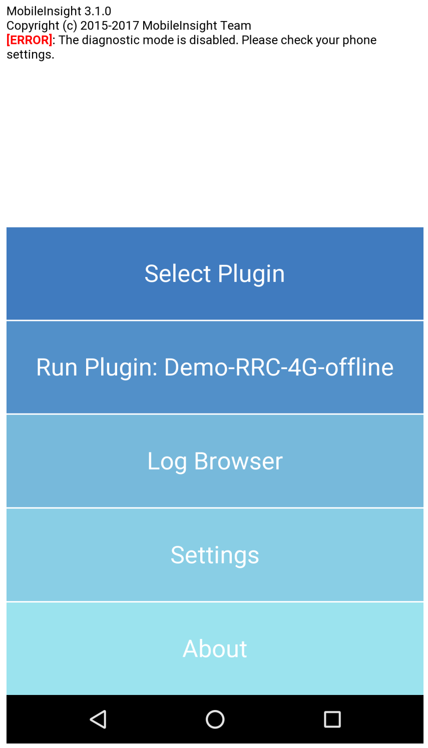

MobileInsight 3.1.0
Copyright (c) 2015-2017 MobileInsight Team
[ERROR]: The diagnostic mode is disabled. Please check your phone settings.

Figure 3: Screenshot of MobileInsight Homepage

order to understand the behavior of Wi-Fi and its interplay with cellular networks for further research and development is crucial for improving upon existing systems. We want to extract low-level messages that are sent to manage Wi-Fi connection as well as collect statistics. For example, we will measure the latency between requesting an association with a base station and the time it takes to complete that association. Another measurement will be to quantify the time it takes for hand-off events to complete i.e. moving from one base station to another. A possible application for this could be, as mentioned before, MPTCP where using measurements from the study of network analytics could be incredibly helpful to researchers and developers. This is because MPTCP is capable of using both cellular and Wi-Fi or the optimal one at a certain time so being able to choose the best network to use can only be done through low-level analytics.

The problem will be solved through a plug-and-play plug-in written using Python for MobileInsight. Developers and researchers using MobileInsight can simply choose to run our plug-in, and network analytics and measurements will be shown on screen and written to a log for further analysis. Thus, by integrating Wi-Fi analytics into Mobile Insight, we are providing a one stop shop for all wireless network analytics, allowing researchers to focus on higher-level issues.

3 APPROACHES

3.1 Overview

We tried and went through several approaches and steps to achieve our goal. Some of them worked but a few did not. We have highlighted all ideas we tried, both successful and unsuccessful, to show the steps we took to come to our solution.

3.2 adb logcat

Firstly, we extracted Wi-Fi related information through the Android system logs (by calling adb logcat on Linux). This first step was to get a feel for the type of information that can be extracted through Android. However, logcat by itself displays a large amount of information from a myriad of Android services. Filtering out the logs to look for those generated by `wpa_supplicant` helped narrow down our focus onto relevant information. `Wpa_supplicant` is a cross platform supplicant that supports WEP, WPA, and WPA2 through which messages being exchanged to manage Wi-Fi connections can be seen. By examining the filtered logs, we were able to get a better grasp of the type of information present and missing in order to formulate our path forward.

3.3 Plugin

After evaluating the logs, we started the process of developing the plug-in for MobileInsight using Python. We began by trying to call the native Android API through python using the python-for-android pyjnius module. Next, we created the WifiManager class from the Android API. The WifiManager class is able to provide coarse grained information about the Wi-Fi Network. Through WifiManager, we can view the currently active Wi-Fi network, establish or tear down connections, and query the dynamic information about the state of the network. Dynamic information about the state of the network is obtained by calling the `getConnectionInfo()` method of the WifiManager class. This will return a WifiInfo object. The WifiInfo object contains methods to return such information like the SSID, frequency, IP address, MAC address, link speed, RSSI, and the state of the current network connection (e.g. authenticating, blocked, connected, disconnected, scanning). However, a lot of the information here was simplistic such as link speed and SSID and we wanted to get more fine-grained information. Thus, as mentioned before, we used `wpa_supplicant` to extract more relevant information.Finally, we created a parser to filter out the relevant information from the extraneous information. The parser can filter out by keywords, making it flexible for the developer so they can tailor the plug-in for their own purposes. The information would be displayed on the screen as well as being saved to a log file.

3.4 Tcpdump and PCAP

Some other approaches we tried include using tcpdump and PCAP. Tcpdump allows us to capture the packets being exchanged but at a higher level. PCAP allows developers to see low-level information such as RTS and CTS messages as well as other control plane information that would have been useful for our project. However, for PCAP, specialized USB Wi-Fi dongles are required in order to capture low-level packets, making it infeasible for our purposes as we did not have the hardware. In addition, tcpdump did not show

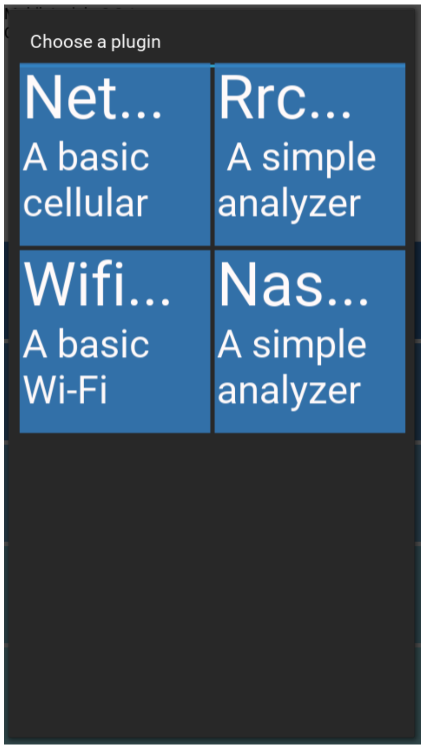


Figure 4: Our plug-in(bottom-left) in plug-in selection screen of MobileInsight

low-level information at the PHY and MAC layer. It showed that Wi-Fi frames were being exchanged but nothing beyond that so it was not very useful for our purposes as we want to know the types of Wi-Fi messages being exchanged.

3.5 Testing

We tested our plug-in in different Wi-Fi usage scenarios such as browsing webpages, online video streaming, and even with Wi-Fi off. We expected different information for the different network use cases as they way the network is being utilized is different. Also, we tested by walking around campus and in buildings in order to model the dynamic connectivity of Wi-Fi networks. As a lot of people use their phones while moving, introducing mobility was necessary for a realistic scenario.

3.6 Statistics

After gathering logs, the next step was to conduct statistical analysis to better understand the data and quantify the latency of operations like handover and the time it takes for association events to complete. Gathering statistics would better allow developers to understand where latency is coming from and the delays associated with different events. In particular, our approach was to time how long it takes to connect to a network after first noticing it as well as time how long hand-off events take.

3.7 Presentation

The last step was to present the information to the user in a readable way. Internally, we have a WifiStatus class that stores the current status of the network in terms of whether it is connected, disconnected, associating, etc. Using this class, we can present relevant information to the user.

4 IMPLEMENTATION

4.1 Overview

We developed a plugin for MobileInsight. As required by the MobileInsight platform, we implemented our plugin in Python, via the Kivy python-for-android framework. We gathered Wi-Fi information through the WifiManager API, which is provided by Android, as well as through wpa_supplicant, a network supplicant that is pre-installed on Android devices that is capable of generating Wi-Fi status information.

Our implementation includes two parts. First, we obtain the information through these two sources in the Python plug-in. Then, we parse this information and present it to the user in an easy-to-read way.

4.2 Tools

We utilize several different software tools made by other researchers or developers. They are listed below.

4.2.1 Kivy

The Kivy python-for-android framework is a package that can pack python code into an Android apk application. With kivy, developers can easily develop Android applications in Python. It also supports accessing Android's Java APIs directly in Python code[9]. It provides the simplicity of developing in Python without losing any functionality provided by the Android Java SDK.

Due to these benefits, MobileInsight is able to support third-party plugins that are written in Python. We use this framework to develop our own plugin.

4.2.2 WifiManager

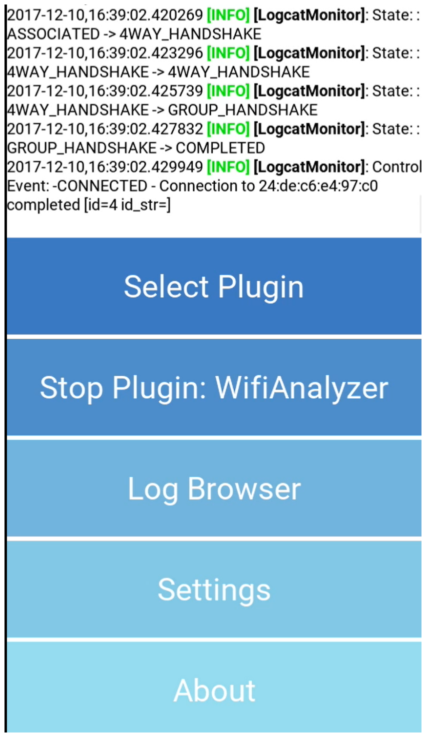
WifiManager is a built-in service in Android. This API provides the functionality of managing all aspects of the Wi-Fi network on the device[1].

In our project, we use this API to get the WifiInfo object. WifiInfo is a class that contains basic information about the current status of the Wi-Fi network. The information provided by WifiInfo includes SSID, IP address, link speed, supplicant speed, and et cetera. As pointed above, we found this information useful in giving the users a basic overview of the network.

4.2.3 wpa_supplicant

Wpa supplicant is a cross-platform wireless supplicant that is installed on Android devices. While its the main functionality is to act as a supplicant and handle roaming as well as authentication/association for the wireless driver, it can also provides users with information about the state of the Wi-Fi network[6].

On Android, wpa supplicant outputs WiFi status information to Android logs, which can be retrieved by running the `adb logcat` command and searching for the keyword `wpa_supplicant` in all the logs obtained from logcat.



```
2017-12-10,16:39:02.420269 [INFO] [LogcatMonitor]: State: :
ASSOCIATED -> 4WAY_HANDSHAKE
2017-12-10,16:39:02.423296 [INFO] [LogcatMonitor]: State: :
4WAY_HANDSHAKE -> 4WAY_HANDSHAKE
2017-12-10,16:39:02.425739 [INFO] [LogcatMonitor]: State: :
4WAY_HANDSHAKE -> GROUP_HANDSHAKE
2017-12-10,16:39:02.427832 [INFO] [LogcatMonitor]: State: :
GROUP_HANDSHAKE -> COMPLETED
2017-12-10,16:39:02.429949 [INFO] [LogcatMonitor]: Control
Event: -CONNECTED - Connection to 24:de:c6:e4:97:c0
completed [id=4 id_str=]
```

Figure 5: Our plug-in running

4.3 Obtain Information

We can obtain the information from the WifiManager API directly by accessing the API in our Python code through the kivy python-for-android framework. It works the same way as what we usually do to access the APIs in an Android application written in Java.

To obtain information from `wpa_supplicant`, things become more complicated. `Wpa_supplicant` will write its output to Android logs, so what we need to do is find a way to access the Android logs through Python code.

4.3.1 Possible Solutions

We came up with several different approaches to obtain this information and we used the most easy and efficient solution in the end. Below are the details about each of the different approaches we considered to retrieve the logs from our Python plug-in.

4.3.1.1 Run Shell Command Through Python

Since wpa_supplicant write its output to Android logs, the first possible approach we could do is directly run the shell command in our Python plug-in to get all the logs. After that, we filter the logs by its origin to obtain only the logs that come from wpa_supplicant.

4.3.1.2 Output to a file

We could also modify the wpa_supplicant's source code. In this approach, we could make wpa_supplicant write output to a file in addition to the Android log. After that, we would be able to easily read the contents of the file through our Python code.

4.3.1.3 Run Shell Command Through C++

In addition, we could also write a small native program in C++. Instead of running shell commands inside the plug-in, we run the commands through the C++ program we call through the plug-in. The C++ program can write the results to a file, and then we read the file through our plug-in written in Python.

4.3.2 Our Solution

We think the first solution, directly running shell command through Python plug-in, is the most efficient solution because there is no middle-ware such as an additional C++ program involved. However, we were having trouble implementing that in the beginning, so we came up with the two additional solutions as alternatives. Eventually, we solved the problems we had, so we returned to our first solution. We have an infinite loop that keeps on executing shell commands to constantly obtain Android logs that are generated by wpa_supplicant. The following pseudo-code shows the procedure to obtain output from wpa_supplicant:

```
while (True) {
    logs = RunShellCommand("logcat -d -s
        wpa_supplicant");
    RunShellCommand("logcat -c");
}
```

What `logcat -d` does is to return logs that are generated by wpa_supplicant only, and with `logcat -c` we clear the cached `logcat` logs so that we do not get duplicate log entries.

4.4 Parse Information

After obtaining all the information we need, the next step is to present users with meaningful output. To achieve this goal, we developed a parser module in the plug-in. We redirect all the logs we gathered from running the shell commands to the parser, and the parser will output some meaningful sentence that explains the current status of the Wi-Fi network. After adding the parser, our procedure looks like the following:

```
while (True) {
    logs = RunShellCommand("logcat -d -s
        wpa_supplicant");
    output = Parse(logs);
    print(output);
    RunShellCommand("logcat -c");
}
```

What the parser does is look for pre-defined keywords in the logs. We have defined some search terms and we only consider log messages containing those search terms as important. We then parse the important log messages to strip away the useless information in it. We then gather the new status of the network and some strings explaining the status in the log. We reconstruct a well-formatted string and present it to the users.

Since the parser is keyword-based, it provides flexibility for future development on this project. If we can manage to make modifications to wpa_supplicant to make it output more useful information, or in case wpa_supplicant behaves differently on a different device, we can easily modify the parser to support new types of output by changing the keyword set.

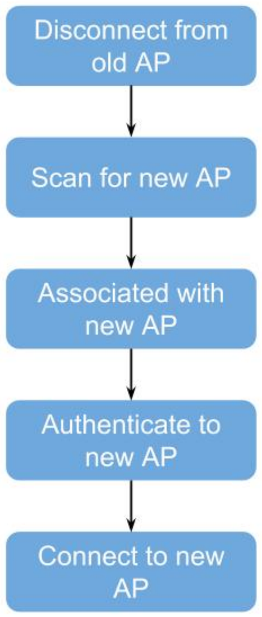
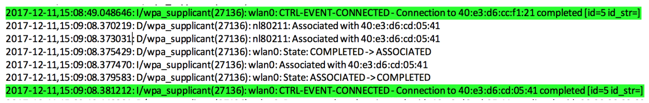


Figure 6: Wi-Fi Finite State Machine

5 RESULTS AND EVALUATIONS

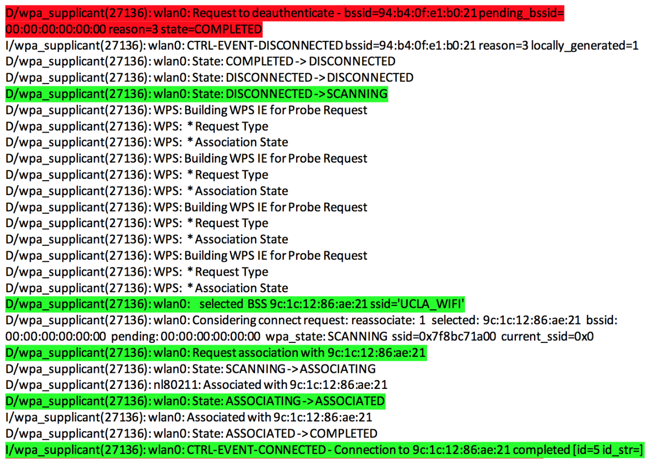
In this project, we used Python to build a runtime Wi-Fi event logger consisting of three parts: a passive monitor on the unsolicited low level event messages sent by wpa_supplicant, a parser with a configurable key words list to extract specific information related to Wi-Fi control events, and an offline analyzer that takes the filtered log as inputs and extracts various Wi-Fi control events, such as association and authentication. In this section, we describe a script that infers runtime WiFi handoff operations while roaming to illustrate how our plugin successfully extracts and analyzes runtime, fine-grained Wi-Fi information.

In this example, our plugin is deployed on a ZTE Obsidian z820, and Wi-Fi events are captured under representative scenarios: a mobile user is walking around randomly in area where Wi-Fi coverage is jointly provided by multiple BSSes, and hence, handoff procedures and Wi-Fi state transitions happen frequently to generate enriched information. These handoff procedures are regulated by management-plane messages which can be readily recorded by our plugin with a key word list containing ﬁProbe Requestﬁ, ﬁAuthenticationﬁ, ﬁAssociation Requestﬁ, etc. The filtered log output from our experiment can be modeled using a finite-state machine as depicted in Figure 6. Clearly, 2 different types of Wi-Fi control events can be identified from these log outputs, namely intra-BSS handoff and recovery from disconnection. Figure 7 presents a representative log message which we use to identify intra-BSS handoff. These log messages reveal that during intra-BSS handoff, mobile device is reassociated to another new access point without explicit disconnection from the old one, and no reassociation request is



```
2017-12-11,15:08:49.048646: I/wpa_supplicant(27136): wlan0: CTRL-EVENT-CONNECTED - Connection to 40:e3:d6:cc:f1:21 completed [id=5 id_str=]
2017-12-11,15:09:08.370219: D/wpa_supplicant(27136): nl80211: Associated with 40:e3:d6:cd:05:41
2017-12-11,15:09:08.373031: D/wpa_supplicant(27136): nl80211: Associated with 40:e3:d6:cd:05:41
2017-12-11,15:09:08.375429: D/wpa_supplicant(27136): wlan0: State: COMPLETED -> ASSOCIATED
2017-12-11,15:09:08.377470: I/wpa_supplicant(27136): wlan0: Associated with 40:e3:d6:cd:05:41
2017-12-11,15:09:08.379583: D/wpa_supplicant(27136): wlan0: State: ASSOCIATED -> COMPLETED
2017-12-11,15:09:08.381212: I/wpa_supplicant(27136): wlan0: CTRL-EVENT-CONNECTED - Connection to 40:e3:d6:cd:05:41 completed [id=5 id_str=]
```

Figure 7: Log messages associated with intra-BSS handoff



```
D/wpa_supplicant(27136): wlan0: Request to deauthenticate - bssid=94:b4:0f:e1:b0:21 pending_bssid=
00:00:00:00:00:00 reason=3 state=COMPLETED
I/wpa_supplicant(27136): wlan0: CTRL-EVENT-DISCONNECTED bssid=94:b4:0f:e1:b0:21 reason=3 locally_generated=1
D/wpa_supplicant(27136): wlan0: State: COMPLETED -> DISCONNECTED
D/wpa_supplicant(27136): wlan0: State: DISCONNECTED -> DISCONNECTED
D/wpa_supplicant(27136): wlan0: State: DISCONNECTED -> SCANNING
D/wpa_supplicant(27136): WPS: Building WPS IE for Probe Request
D/wpa_supplicant(27136): WPS:  * Request Type
D/wpa_supplicant(27136): WPS:  * Association State
D/wpa_supplicant(27136): WPS: Building WPS IE for Probe Request
D/wpa_supplicant(27136): WPS:  * Request Type
D/wpa_supplicant(27136): WPS:  * Association State
D/wpa_supplicant(27136): WPS: Building WPS IE for Probe Request
D/wpa_supplicant(27136): WPS:  * Request Type
D/wpa_supplicant(27136): WPS:  * Association State
D/wpa_supplicant(27136): WPS: Building WPS IE for Probe Request
D/wpa_supplicant(27136): WPS:  * Request Type
D/wpa_supplicant(27136): WPS:  * Association State
D/wpa_supplicant(27136): wlan0:   selected BSS 9c:1c:12:86:ae:21 ssid='UCLA_WIFI'
D/wpa_supplicant(27136): wlan0: Considering connect request: reassociate: 1  selected: 9c:1c:12:86:ae:21  bssid:
00:00:00:00:00:00  pending: 00:00:00:00:00:00  wpa_state: SCANNING ssid=0x7f8bc71a00 current_ssid=0x0
D/wpa_supplicant(27136): wlan0: Request association with 9c:1c:12:86:ae:21
D/wpa_supplicant(27136): wlan0: State: SCANNING -> ASSOCIATING
D/wpa_supplicant(27136): nl80211: Associated with 9c:1c:12:86:ae:21
D/wpa_supplicant(27136): wlan0: State: ASSOCIATING -> ASSOCIATED
I/wpa_supplicant(27136): wlan0: Associated with 9c:1c:12:86:ae:21
D/wpa_supplicant(27136): wlan0: State: ASSOCIATED -> COMPLETED
I/wpa_supplicant(27136): wlan0: CTRL-EVENT-CONNECTED - Connection to 9c:1c:12:86:ae:21 completed [id=5 id_str=]
```

Figure 8: Log messages associated with recovery from disconnection

sent from client side. Therefore, it can be inferred that intra-BSS handoff is triggered by the network-side or Android OS. This event is more likely to happen when mobile user is roaming in an area where Wi-Fi coverage is seamlessly provided by multiple access points of the same BSS. Obviously, intra-BSS handoff doesn't go through every procedure as depicted in Figure 6, which results in a shorter completion time.

On the other hand, another message pattern, which is significantly different from that of intra-BSS handoff, is also observed as shown in Figure 8. Compared to intra-BSS handoff, the mobile device undergoes every procedure in the Wi-Fi finite-state machine: When the user is leaving (or has left) current BSS, the mobile device detects a poor link to its current access point, and take the initiative to disconnect from current access point soon afterwards. Then it uses an active scanning function to find and send an association request to another access point. Noteworthy, even though the mobile device may again attach to the previous BSS shortly, it always goes through a disconnection. Consequently, we are unable to distinguish whether the mobile device is out of the Wi-Fi coverage or just smoothly migrates to another BSS. It should be noted we also observed repeated log entries indicating a probing event/re-association request is sent during this process suggesting multiple failed attempts and long completion time.

These parsed messages are then fed into the analyzer component to extract the timestamp as well as a completion time for every handoff event and to calculate the statistics. Figure 9 depicts the trace of these control events during our test, and their associated execution times. It should be pointed out that the execution time of intra-BSS is slightly underestimated because such event is initiated by the network side and our plugin has no direct access to the network-side states and events to accurately calculate the total association time. At Time = 0s, we switch on the Wi-Fi on our mobile devices, and our test ends at Time = 900s. During this period of time, 10 recoveries from disconnection events and 15 intra-BSS handoff events occur. The missing datapoints during Time = 450s $\sim$ 550s imply the mobile device remains stationary and no control event is triggered. Note that, this example is not intended to be complete. Instead, it serves to demonstrate the effectiveness and usefulness of re-constructing and analyzing Wi-Fi control events with our plugin. As expected, intra-BSS handoff happens more frequently than recovery from disconnection, and the time required for reassociation to a new access point is orders of magnitudes smaller compared to recovery from disconnection. However, at Time = 270s, a high association time for intra-BSS handoff is observed, and the associated log message reveals that multiple failed attempts to reassociate could also occur during intra-BSS handoff. We summarize several basic statistics for both events in Table 1. Given that the median is significantly smaller than mean for both control events, we can conclude that most events have small completion time while the large mean values are attributed to a small amount of anomalous events and outliers.

Next, we wanted to find out the causes of the prolonged completion time and large deviation of recovery for disconnecction events. Our plugin also yields detailed analysis on each procedure happened during control event as presented in Figure 10. Clearly, for most of the recovery events, the authentication takes only a few milliseconds. While most of the completion time is spend on either scanning or association, because when mobile devices perform scanning and association, no radio resource is reserved and it has no information on the states of access points. This information is critical for mobile devices to make optimal decisions on scanning or association, without which, repeated scanning or association events may occur, and network variance will induce large deviation of completion time of scanning or association. Indeed, such events are recorded in our log outputs as shown in Figure 11. Once association is completed, the radio quality is more likely to be good, which leads to a short completion time of authentication. Such reasoning is also consistent with the finding that for most recovery events, at most one procedure has prolonged completion time. With these functionalities, our plugin can offer critical information to analyzes runtime, fine-grained Wi-Fi events.

Table 1: Completion Time Statistic

	Intra-BSS Handoff	Recovery from Disconnection
Number	15	10
Mean	4.32ms	3272.33ms
Median	3.77ms	1411.69ms
Deviation	1.85ms	6331.89ms
Minimum	2.94ms	29.08ms
Maximum	10.80ms	21995.85ms

6 DIFFICULTIES AND SOLUTIONS

6.1 Wi-Fi Information

We are quite limited in what information we can gather with just the phone alone. The project specs asked us to examine link layer information to see if we can maybe extract information such as RTS/CTS packets, channel estimation, modulation and coding scheme. However, it is not possible to capture raw 802.11 frames from the internal device NIC without custom firmware and drivers [3]. A

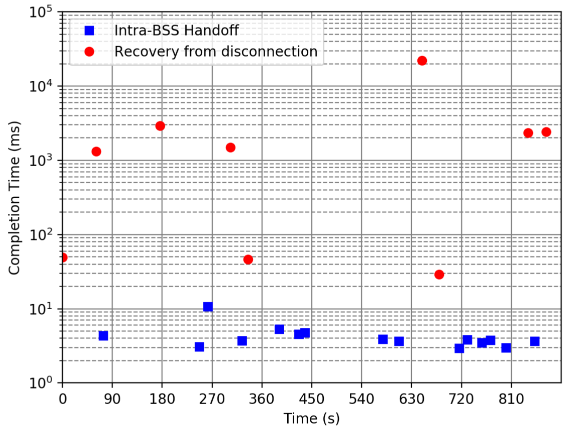


Figure 9: Runtime Handoff Event Completion Times

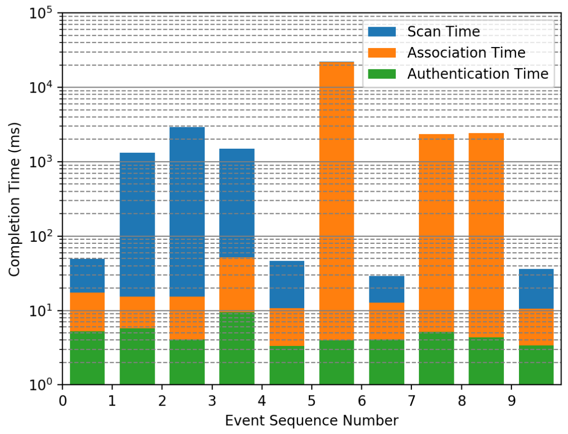
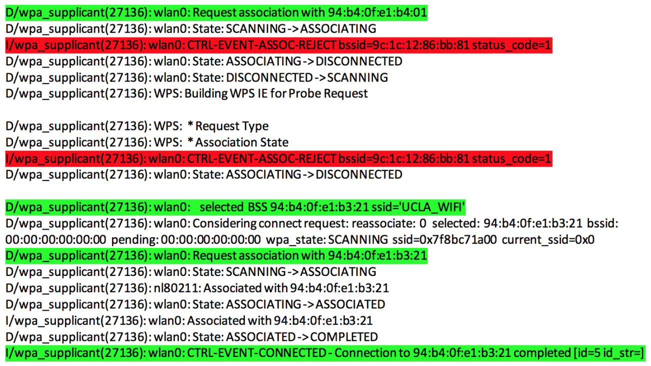


Figure 10: Completion Times of Different Procedures in Handoff Event



```
D/wpa_supplicant(27136): wlan0: Request association with 94:b4:0f:e1:b4:01
D/wpa_supplicant(27136): wlan0: State: SCANNING -> ASSOCIATING
I/wpa_supplicant(27136): wlan0: CTRL-EVENT-ASSOC-REJECT bssid=9c:1c:12:86:bb:81 status_code=1
D/wpa_supplicant(27136): wlan0: State: ASSOCIATING -> DISCONNECTED
D/wpa_supplicant(27136): wlan0: State: DISCONNECTED -> SCANNING
D/wpa_supplicant(27136): WPS: Building WPS IE for Probe Request

D/wpa_supplicant(27136): WPS:  * Request Type
D/wpa_supplicant(27136): WPS:  * Association State
I/wpa_supplicant(27136): wlan0: CTRL-EVENT-ASSOC-REJECT bssid=9c:1c:12:86:bb:81 status_code=1
D/wpa_supplicant(27136): wlan0: State: ASSOCIATING -> DISCONNECTED

D/wpa_supplicant(27136): wlan0:   selected BSS 94:b4:0f:e1:b3:21 ssid='UCLA_WIFI'
D/wpa_supplicant(27136): wlan0: Considering connect request: reassociate: 0  selected: 94:b4:0f:e1:b3:21 bssid:
00:00:00:00:00:00  pending: 00:00:00:00:00:00  wpa_state: SCANNING ssid=0x7f8bc71a00  current_ssid=0x0
D/wpa_supplicant(27136): wlan0: Request association with 94:b4:0f:e1:b3:21
D/wpa_supplicant(27136): wlan0: State: SCANNING -> ASSOCIATING
D/wpa_supplicant(27136): nl80211: Associated with 94:b4:0f:e1:b3:21
D/wpa_supplicant(27136): wlan0: State: ASSOCIATING -> ASSOCIATED
I/wpa_supplicant(27136): wlan0: Associated with 94:b4:0f:e1:b3:21
D/wpa_supplicant(27136): wlan0: State: ASSOCIATED -> COMPLETED
I/wpa_supplicant(27136): wlan0: CTRL-EVENT-CONNECTED - Connection to 94:b4:0f:e1:b3:21 completed [id=5 id_str=]
```

Figure 11: Log messages associated with association failures

workaround exists where a separate Wi-Fi adapter can be connected via USB-OTG, but we did not have the hardware to explore that approach. As such, collecting link layer information was delegated as a future improvement task.

6.2 Mobile Insight Plug-in Interface

Coming into this project, there were a lot of difficulties working with the Mobile Insight plug-in interface. We were not very familiar with the architecture of the app and things that seemed like they should work did not work. The documentation for the app and its functions are also very sparse. As a result, it was very difficult for us to debug simple problems, and it took a lot of help from the TAs to figure things out.

One of the first issues we ran into was with the python for android API. We wanted to extract coarse-grain Wi-Fi information by calling native android APIs. To do so, the android developer reference tells us we need to obtain an instance of the WifiManager class by calling

```
Context.getSystemService(Context.
    WIFI_SERVICE)
```

In the python for android API, the methodology as shown in the documentation is

```
PythonActivity = autoclass('org.kivy.android
    .PythonActivity')
activity = PythonActivity.mActivity
Context = autoclass('android.content.Context
    ')
WifiManager = activity.getSystemService(
    Context.WIFI_SERVICE)
```

This, however, did not work in the Mobile Insight plug-in interface. As it turns out, in order to properly get an instance of the class, we have to call

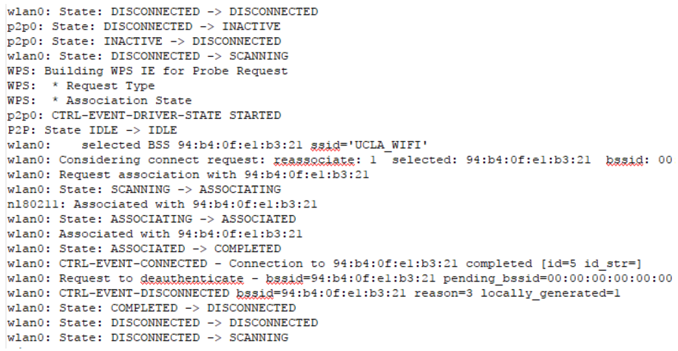
```
PythonService = autoclass('org.kivy.android.
    PythonService')
service = PythonService.mService
Context = autoclass('android.content.Context
    ')
WifiManager = activity.getSystemService(
    Context.WIFI_SERVICE)
```

This piece of information was buried as one line in the old documentation for python for android, and never mentioned in Mobile Insight documentation. We would have never figured that out without TA help.

We had a lot of difficulty trying to figure out to how to run shell commands through the plug-in interface. Our goal was simple: try to call logcat and pipe the output to grep in order to look for output from "wpa_supplicant." Our initial method worked when we ran it using python from the adb shell command line interface. However, we were never able to get this to work on the Mobile Insight python plug-in interface.

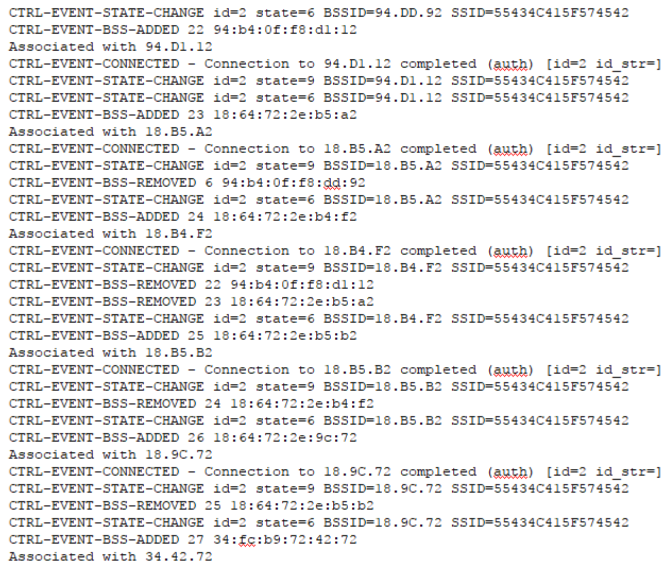
```
process = subprocess.Popen(['logcat', '|', '
    grep', 'wpa_supplicant'], stdout=
    subprocess.PIPE)
output = process.stdout.readline()
```

With help from the TA, we were instead able to get the "wpa_supplicant" log output directly without calling for grep by searching for the



```
wlan0: State: DISCONNECTED -> DISCONNECTED
p2p0: State: DISCONNECTED -> INACTIVE
p2p0: State: INACTIVE -> DISCONNECTED
wlan0: State: DISCONNECTED -> SCANNING
WPS: Building WPS IE for Probe Request
WPS:  * Request Type
WPS:  * Association State
p2p0: CTRL-EVENT-DRIVER-STATE STARTED
P2P: State IDLE -> IDLE
wlan0:    selected BSS 94:b4:0f:e1:b3:21 ssid='UCLA_WIFI'
wlan0: Considering connect request: reassociate: 1  selected: 94:b4:0f:e1:b3:21  bssid: 00
wlan0: Request association with 94:b4:0f:e1:b3:21
wlan0: State: SCANNING -> ASSOCIATING
nl80211: Associated with 94:b4:0f:e1:b3:21
wlan0: State: ASSOCIATING -> ASSOCIATED
wlan0: Associated with 94:b4:0f:e1:b3:21
wlan0: State: ASSOCIATED -> COMPLETED
wlan0: CTRL-EVENT-CONNECTED - Connection to 94:b4:0f:e1:b3:21 completed [id=5 id_str=]
wlan0: Request to deauthenticate - bssid=94:b4:0f:e1:b3:21 pending_bssid=00:00:00:00:00:00
wlan0: CTRL-EVENT-DISCONNECTED bssid=94:b4:0f:e1:b3:21 reason=3 locally_generated=1
wlan0: State: COMPLETED -> DISCONNECTED
wlan0: State: DISCONNECTED -> DISCONNECTED
wlan0: State: DISCONNECTED -> SCANNING
```

Figure 12: Parsed log output from the ZTE Obsidian z820



```
CTRL-EVENT-STATE-CHANGE id=2 state=6 BSSID=94.DD.92 SSID=55434C415F574542
CTRL-EVENT-BSS-ADDED 22 94:b4:0f:f8:d1:12
Associated with 94.D1.12
CTRL-EVENT-CONNECTED - Connection to 94.D1.12 completed (auth) [id=2 id_str=]
CTRL-EVENT-STATE-CHANGE id=2 state=9 BSSID=94.D1.12 SSID=55434C415F574542
CTRL-EVENT-STATE-CHANGE id=2 state=6 BSSID=94.D1.12 SSID=55434C415F574542
CTRL-EVENT-BSS-ADDED 23 18:64:72:2e:b5:a2
Associated with 18.B5.A2
CTRL-EVENT-CONNECTED - Connection to 18.B5.A2 completed (auth) [id=2 id_str=]
CTRL-EVENT-STATE-CHANGE id=2 state=9 BSSID=18.B5.A2 SSID=55434C415F574542
CTRL-EVENT-BSS-REMOVED 6 94:b4:0f:f8:dd:92
CTRL-EVENT-STATE-CHANGE id=2 state=6 BSSID=18.B5.A2 SSID=55434C415F574542
CTRL-EVENT-BSS-ADDED 24 18:64:72:2e:b4:f2
Associated with 18.B4.F2
CTRL-EVENT-CONNECTED - Connection to 18.B4.F2 completed (auth) [id=2 id_str=]
CTRL-EVENT-STATE-CHANGE id=2 state=9 BSSID=18.B4.F2 SSID=55434C415F574542
CTRL-EVENT-BSS-REMOVED 22 94:b4:0f:f8:d1:12
CTRL-EVENT-BSS-REMOVED 23 18:64:72:2e:b5:a2
CTRL-EVENT-STATE-CHANGE id=2 state=6 BSSID=18.B4.F2 SSID=55434C415F574542
CTRL-EVENT-BSS-ADDED 25 18:64:72:2e:b5:b2
Associated with 18.B5.B2
CTRL-EVENT-CONNECTED - Connection to 18.B5.B2 completed (auth) [id=2 id_str=]
CTRL-EVENT-STATE-CHANGE id=2 state=9 BSSID=18.B5.B2 SSID=55434C415F574542
CTRL-EVENT-BSS-REMOVED 24 18:64:72:2e:b4:f2
CTRL-EVENT-STATE-CHANGE id=2 state=6 BSSID=18.B5.B2 SSID=55434C415F574542
CTRL-EVENT-BSS-ADDED 26 18:64:72:2e:9c:72
Associated with 18.9C.72
CTRL-EVENT-CONNECTED - Connection to 18.9C.72 completed (auth) [id=2 id_str=]
CTRL-EVENT-STATE-CHANGE id=2 state=9 BSSID=18.9C.72 SSID=55434C415F574542
CTRL-EVENT-BSS-REMOVED 25 18:64:72:2e:b5:b2
CTRL-EVENT-STATE-CHANGE id=2 state=6 BSSID=18.9C.72 SSID=55434C415F574542
CTRL-EVENT-BSS-ADDED 27 34:fc:b9:72:42:72
Associated with 34.42.72
```

Figure 13: Parsed log output from the Samsung Galaxy Note 3

”wpa_supplicant” PID beforehand and passing that in as a parameter to logcat.

```
lines = util.run_shell_cmd("logcat --pid %s
    -d" % self.ps_num, True).split('\n')
```

Later on we discovered there was a much simpler method that worked and that was to call the -s option of logcat.

```
lines = util.run_shell_cmd("logcat -d -s
    wpa_supplicant", True).split('\n')
```

We are still puzzled why our original solution doesn't work within Mobile Insight when it works fine outside. A lot of development time was spent trying to resolve these discrepancies.

6.3 wpa_supplicant and Unification

Our development was primarily done on one device: a ZTE Obsidian z820. During testing, we also tried running the plug-in on one of our group member's Samsung Galaxy Note 3. We discovered that the logs produced by the two mobile devices were not the same and that the keywords we were using for the ZTE did not work on the Note 3. We suspect that the Note 3 has a different version of ”wpa_supplicant” installed, so different things are printed out to its log.

Figures 12 and 13 show the log output from the ZTE and the Note 3 respectively using the same parser with the same keywords. Note how the log outputs are very different from each other.

```
wlan0: Request association with 94:b4:0f:e1:
    b3:21
wlan0: State: SCANNING -> ASSOCIATING
nl80211: Associated with 94:b4:0f:e1:b3:21
wlan0: State: ASSOCIATING -> ASSOCIATED
wlan0: Associated with 94:b4:0f:e1:b3:21
wlan0: State: ASSOCIATED -> COMPLETED
wlan0: CTRL-EVENT-CONNECTED - Connection to
    94:b4:0f:e1:b3:21 completed [id=5 id_str
    =]
```

With the Note 3, these more detailed state change outputs are missing as shown in the snippet below:

```
CTRL-EVENT-CONNECTED - Connection to 18.B4.
    F2 completed (auth) [id=2 id_str=]
CTRL-EVENT-STATE-CHANGE id=2 state=9 BSSID
    =18.B4.F2 SSID=55434C415F574542
CTRL-EVENT-BSS-REMOVED 22 94:b4:0f:f8:d1:12
CTRL-EVENT-BSS-REMOVED 23 18:64:72:2e:b5:a2
CTRL-EVENT-STATE-CHANGE id=2 state=6 BSSID
    =18.B4.F2 SSID=55434C415F574542
CTRL-EVENT-BSS-ADDED 25 18:64:72:2e:b5:b2
Associated with 18.B5.B2
```

However, we did observe similar finite states in those logs. We concluded that the logs gave use similar information, but in a different format. Since we have provided flexibility in our parse implementation, this problem could be solved modifying the keywords used in our parser. For each device, we could first manually determine the keyword to use on the specific version of wpa_supplicant by either browsing all the logs or the source code of wpa_supplicant.

Another solution is modifying the wpa_supplicant on each device to output the same logs on the same events. However, this requires more modification to the phone and it may be difficult for users to deploy those changes in order to run our plug-in. Therefore, we believe the first solution to modify the keywords based on each device is better and easier to implement. We may store the different keywords for different devices in a file as a part of our plug-in, and use Android APIs to get the model of the device in the beginning so that we can use the correct keywords.

Since we have limited access to different models of the phones, we were unable to provide our final solution to work on all the devices, but we did come up with the potential solution and the change will be easy to make.

7 FUTURE WORK

The availability of low level Wi-Fi information given just a phone is very low. All smartphones have an internal Wi-Fi NIC, but it is not possible to capture raw frames from the internal Wi-Fi interface without custom firmware and root access [3]. To get around this, Kismet Wireless has produced a utility called Android PCAP which can capture raw 802.11 frames in ”monitor mode”, sometimes referred to as ”promiscuous mode.” This works by capturing frames not from the internal NIC, but from an external USB Wi-Fi adapter connected over USB-OTG. This way, we can extract low level Wi-Fi information down to the link layer which we do not have with wpa_supplicant alone.

Since we are limited by the amount of information given out by wpa_supplicant, another avenue for future work could involve modifying wpa_supplicant to print out more information, or printing out information that is more user friendly to read. Moreover, having a consistent version of wpa_supplicant across devices with consistent log messages printed would alleviate the problem of different mobile devices printing out different things.

Another avenue for improvement and future work is in how the information is displayed and what information to display. Currently what we have done is filter out a lot of extraneous log output and display only those messages pertinent to connection state and connection hand off. We can walk around with our mobile device and see what our Wi-Fi connection state is and when hand offs occur. However, what the user sees are still log messages which may still be too verbose and not too user-friendly. The verbosity may be a problem for the current mobile insight interface where messages are printed out to a small window at the top half of the screen. It is not able to contain many messages, and it definitely cannot handle cases where a lot of messages arrive in a short period of time. A possible way to approach this is to cut down the messages and further parse them down to a set of key words or states.

Since the logs are saved, it is possible to post process them to obtain more information - namely we can get statistics about the events that have occurred using the log messages and time stamps. The logs can be post processed in another plug-in and statistical information about the events could be returned. Interesting information that could be returned could include run time, total number of events, number of association/disassociation events, average time for hand-offs, and average time between hand-offs. This may be difficult since the events printed out for each phone is different depending on the wpa_supplicant installed. This is where modifying wpa_supplicant or further log parsing may come in to ensure event messages are similar between devices.

In addition, as mentioned before, our solution was tested on only one phone model. We addressed possible ways we could solve this issue, but a future work could be to actually implement a working solution. We understand that a solution that only works on one phone model is not very usefl for developers and researchers and we should have thought more about that, but the lack of other phone models to use as well as our lack of thought regarding this issue led to this oversight. Thus, in the future, we could modify the plugin to be able to work on any phone model and making it a much more useful solution.

8 SUMMARY

For our project, we extended the functionalities of MobileInsight, the mobile application that is capable of performing analysis on cellular networks, through a plugin inside the application. Our goal was to make it easier for developers to evaluate and analyze low-level WiFi information which is not as simple as may seem to extract. In our plugin we obtained detailed information about the current status of the WiFi network from the Android API as well as `wpa_supplicant` and we presented this information in MobileInsight and saved it to a log. This information included the types of messages being sent to manage WiFi connections as well such as authentication and association events. All of this information could be useful to developers as it can provide avenue for further analysis and help pinpoint areas of possible improvement in existing protocols or systems. In addition, we analyzed the data we got from the device ourselves and we found out the various latencies associated with different WiFi network events such as the time for associating to a BS, the time for moving from one BSS to another BSS, and the time for moving from one AP to another AP within the same BSS. These timing events demonstrate the dynamics of a variable WiFi connection and can prove useful to developers in understanding the inherent latencies present in wiFi networks. Overall, this project made us understand the semantics and protocols behind WiFi networks more and we now have a better understanding of what goes on behind the scenes to make WiFi so indispensable and set the stage for further research and development in this area.

9 APPENDIX

During the presentation, the professor brought up the question of statistical analytics. In our project thus far, we have filtered down log information to show Wi-Fi state changes and association/disassociation events. However, we have not provided any metrics for these events. To address that, we have written a python script that takes our filtered log as input and outputs a list of relevant metrics. In addition, Zengwen mentioned that we should address the issue regarding our lack of a generic solution as we tested our solution only on one phone. To rectify that, we wrote about possible solutions in section 6.3.

ACKNOWLEDGMENTS

We would like to thank our mentors on this project, Zengwen Yuan and Yuanjie Li. They were extremely helpful and supportive to us and their paper on Mobile Insight proved to be useful in understanding the problem and approach better.

REFERENCES

[1] Android. [n. d.]. WifiManager. https://developer.android.com/reference/android/net/wifi/WifiManager.html. ([n. d.]). Accessed: 12-12-2017.
[2] Cisco. March 2017. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016fi?!2021 White Paper. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-net working-index-vni/mobile-white-paper-c11-520862.html. (March 2017). Accessed: 10-15-2017.
[3] dragorn. Jan 2012. Capturing raw 802.11 on android. http://blog.kismetwireless.net/2012/01/capturing-raw-80211-on-android.html. (Jan 2012). Accessed: 10-14-2017.
[4] Yuanjie Li, Chunyi Peng, Zengwen Yuan, Jiayao Li, Haotian Deng, and Tao Wang. Oct 2016. MobileInsight: Extracting and Analyzing Cellular Network Information on Smartphones. In *Proceedings of the The 22nd Annual International Conference on Mobile Computing and Networking (MobiCom '16)*. ACM.
[5] The Nielsen Company (US) LLC. Nov 2016. WHAT DRIVES DATA USAGE? http://www.nielsen.com/us/en/insights/news/2016/what-drives-data-usage.html. (Nov 2016). Accessed: 12-12-2017.
[6] Jouni Malinen and contributors. 2017. WPA Supplicant. http://w1.fi/cgit/hostap/plain/wpa_supplicant/README. (2017). Accessed: 12-12-2017.
[7] mobidia. [n. d.]. Aggregate Monthly Wi-Fi vs. Cellular Usage for Select Countries. http://mobidia.redlabelcom.com/products/analytics/. ([n. d.]). Accessed: 12-10-2017.

[8] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. Apr 2012. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*. USENIX.

[9] Alexander Taylor. 2015. python-for-android. https://python-for-android.readthedocs.io/en/latest/. (2015). Accessed: 12-12-2017.