Enhancing Mobile Forensics on iOS

by

Jeremy Whitaker

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2015 by the
Graduate Supervisory Committee:

Gail-Joon Ahn, Chair
Adam Doupé
Stephen Yau

ARIZONA STATE UNIVERSITY

December 2015

ABSTRACT

Due to the shortcomings of modern Mobile Device Management solutions, businesses have begun to incorporate forensics to analyze their mobile devices and respond to any incidents of malicious activity in order to protect their sensitive data. Current forensic tools, however, can only look a static image of the device being examined, making it difficult for a forensic analyst to produce conclusive results regarding the integrity of any sensitive data on the device. This research thesis expands on the use of forensics to secure data by implementing an agent on a mobile device that can continually collect information regarding the state of the device. This information is then sent to a separate server in the form of log files to be analyzed using a specialized tool. The analysis tool is able to look at the data collected from the device over time and perform specific calculations, according to the user's specifications, highlighting any correlations or anomalies among the data which might be considered suspicious to a forensic analyst. The contribution of this paper is both an in-depth explanation on the implementation of an iOS application to be used to improve the mobile forensics process as well as a proof-of-concept experiment showing how evidence collected over time can be used to improve the accuracy of a forensic analysis.

To my family and friends

ACKNOWLEDGMENTS

After now having gone through both the undergraduate and Graduate Computer Science programs at Arizona State University, I feel a great sense of accomplishment but also somewhat overwhelmed as I now understand just how vast the field of Computer Science, Information Assurance in particular, actually is. I know that this will be an important milestone in my life, and I would like to express my sincerest gratitude toward Dr. Gail-Joon Ahn who, being both a mentor and a friend to me, has encouraged and directed me throughout every step of my graduate career as well as the latter parts of my undergraduate career. I would also like to thank my colleagues at the Secure Engineering for Future Computing (SEFCOM) laboratory, especially Michael Mabey, for helping to guide me through my graduate degree. Furthermore I would like to thank Dr. Stephen Yau and Dr. Adam Doupé for serving on my committee and evaluating my research. Additionally I would like to to thank the information security team at Freeport McMoRan for providing devices for this research project. Finally I would like to extend my thanks to my friends and family who have been a constant support for me during this period in my career.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Mobile devices have become a critical tool for any modern employee. Companies typically want their employees to quickly communicate information related to their business goals, and as a result each employee will often store sensitive company information on his or her device. Naturally this information needs to be secured, so businesses will often utilize Mobile Device Management (MDM) software to ensure that their sensitive data is protected. One of the key aspects of MDM software is the security features that they offer, namely allowing encryption of sensitive data, remote locking, and remote wiping; however, even with these features in place, attackers have still been able to breach devices that use MDM software through vulnerabilities in the operating system and privilege escalation [1]. Even though no system is ever completely secure, many companies are solely reliant on MDM software for their security needs, and because of this they have no set course of action for when their sensitive information is compromised [2].

Computer Forensics is a discipline that involves obtaining and analyzing computer data for judicial purposes. Typically computer forensics techniques are only used in solving computer-related crimes; however, they can also be used as a possible solution for the lack of intrusion detection measures present in a lot of MDM software. In [3], Scholnick states the use of forensic methods is one of two primary weapons that businesses have for protecting enterprise security because it provides "evidentiary discovery." In other words, companies can use forensic methods to find evidence of malicious activity and then respond once they determine that their sensitive information may have been compromised. Our approach expands on this premise by

implementing a system that would allow for a more accurate assessment of the state of a mobile device.

One common security mechanism of Mobile Device Management tools is to prevent the installation of any third-party application [1]; however, this sort-of defense mechanism typically relies on signature verification and can be bypassed if additional checks are not in place. Such is the case with Gatekeeper program on OSX which protects the system by only allowing programs that are signed by known third-party developers to run. According to an article by Edward Kovacs [4], the Gatekeeper program does not check all included, externally-referenced binaries, which means that an attacker could replace a non-malicious binary, that's being externally-referenced by a signed application, with a malicious binary, and the signed application would still execute it. Because MDM systems use an application validation mechanism similar to that used by the Gatekeeper program for OSX, it follows that a similar attack would be possible on iOS, assuming that an approved application that runs the external binary can be installed on the device. Additionally, due to MDM's lack of intrusion detection, the victims would likely not be able to determine the cause of the attack. Using forensic techniques, however, a user might be able to gather evidence to piece together a story about the origins of any recent malicious activity on the device. In the case of the Gatekeeper vulnerability, for example, a user could use forensic techniques to see which processes had recently been launched and are currently using system memory. He or she could then check to see if any processes had similar launch times, which would indicate that they were related, and from there begin to speculate that a malicious process was launched from a non-malicious one.

Access Data's Mobile Phone Examiner Plus (MPEPlus) is one of the most widely-used mobile forensic tools, and it is a good example of software that a business's security manager could use to extract data from a device to investigate whether or not

the sensitive information on their devices had been compromised. Although the data extraction methods vary depending on what types of operating systems are running on the devices being examined, extracting data using MPEPlus typically works by installing a temporary agent on the mobile device and then transferring all accessible files to a logical image of the device on the host computer which can then be explored using the tool. Once all data has been extracted from the mobile device, a forensic examiner can then search through the various files, logs, and databases in an attempt to find any evidence of malicious activity. For example, if after extracting all data from a mobile device an analyst saw that several unrecognized SMS messages had been sent to an unrecognized number and that an application with an unrecognized publisher was installed on the device, the analyst could conjecture that a malicious application had been installed on the device which was sending SMS messages for nefarious purposes.

Although tools like MPEPlus can be used to help identify whether or not a business's devices have been compromised, and even though the collected evidence may seem somewhat suspicious to an analyst, often times he or she may not have enough information to draw a definitive conclusion about the integrity of the data on the device. When considering the example with the unrecognized application and unrecognized SMS recipient, although the examiner could suspect that the unrecognized application was a malicious application responsible for sending the SMS messages, there is no evidence that those two factors are necessarily related. Another possibility could be that the user simply let someone else use the device to contact someone that the user did not recognize. If this were the case, then there would not be enough data for an analyst to confidently draw a conclusion. If, however, the analyst also had a time-line of events showing that the unrecognized recipient was contacted via SMS at the same times that the unrecognized application was active, then the evi-

dence would be more complete, and the analyst could more confidently conclude that the application may be malicious. Similarly, in the case of bypassing the Gatekeeper program in OSX, although the user can use forensic techniques to speculate that a malicious process may be linked to a non-malicious process, he or she may not have enough information to make that conclusion definitively, but with a time-line of events, the user might be able to see a consistent pattern of a specific process being launched in conjunction with another process.

As described in the paper by Paglierani et al. in any forensic investigation, the investigators should ensure that any collected evidence adheres to the four major rules of evidence [5]:

1. Authenticity — The evidence can be linked to a specific individual or incident.
2. Admissibility — The evidence was legally obtained and can be admitted in a court.
3. Completeness — The evidence is not partial; it includes all available information.
4. Reliability / Accuracy — The evidence collection procedure is explainable.

Our approach focuses primarily on improving the completeness of the evidence that can be collected from a mobile device. In contrast to incomplete evidence, complete evidence is capable of aiding an analyst in drawing conclusions about an event or series of events with a high level of confidence. Although a commercial tool like MPEPlus can certainly offer an analyst insight as to whether or not malicious activity had occurred on a mobile device, the analyst often times is unable to confidently draw conclusions about specific events due to only having glimpses of what took place, i.e. the analyst only has partial evidence. How then can forensics be used to

help an analyst definitively determine the integrity of the data on a mobile device? The answer, as previously mentioned, is that the analyst needs to be able to look at the events that transpire over a period of time. In other words, there needs to be a way to continuously collect information about the state of the device over time and order the events along a time-line which can be forensically examined. Scholnick mentions this in his article when talking about comprehensive monitoring and tracking capabilities [3]. Our approach discussed in this thesis uses similar techniques for enhancing the completeness of the forensic evidence, allowing an analyst to have a more comprehensive understanding of the events that transpired.

It seems somewhat obvious that continuously collecting information is advantageous for an analyst compared to only gaining a snapshot of the information through a commercial tool like MPEPlus; however, there are various restrictions that make this approach challenging. One of the restrictions, which will be discussed in more detail later, is the sandboxing mechanism for the iOS platform which limits our continuous collection of data by allowing us to collect only specific types of information from iOS devices. Another restriction is that with iOS, the kind of methods required to even obtain the limited information available would cause any application in which they are implemented to be rejected from the official application store, making deployment of the application impractical. It is still possible, however, to use a development license to install such an application on actual iOS devices, and this sort of deployment mechanism is much more feasible in a corporation setting, which is the setting from which the motivation for our approach originated. Additionally, although the data we can collect is limited, it is useful in that it still can give an analyst further insight into the events that transpired on the device to some degree if used alongside a commercial forensics tool. Furthermore, our approach not only demonstrates how continual data collection is useful in spite of these restrictions, it also shows specifically how a more

complete data set can be used to improve the accuracy and efficiency of a forensic analysis.

Chapter 2

RELATED WORK

The aim of some of the more recent advancements in mobile forensics techniques is to provide forensic analysts with ways to draw accurate, complete conclusions regarding the events that occurred on the device based on the extracted evidence. As mentioned previously, one of the most recent advancements involves continuously monitoring the mobile device and collecting information from it; however, with this advancement, it is important to know precisely what information to collect. This research thesis focuses primarily on advancing mobile forensics on the iOS platform, so in order to ensure optimal development for this advancement in mobile forensics, we needed to have an understanding of various threats and attacks targeted at the iOS platform in addition to any threats that could impact mobile devices in general. The study of these threats, in addition to our Preliminary Investigation discussed in section 7.1, are what helped us determine what types of information are essential in conducting a mobile forensics investigation.

One type of threat is discussed in [6], the authors describe how users often times inherently trust the host that is providing a mobile device with power via a USB connection, and as a result, make themselves vulnerable to unapproved installation of malicious applications on their devices. By using the same communication protocols as iTunes, the authors were able to obtain the device's UDID using their proof-of-concept malicious charger host. They then created a provisioning profile which they could use to install any iOS application on a victim's device, bypassing the vetting process of the iOS AppStore. Knowing the mechanisms of this type of attack, we need to determine what information we should continually try to obtain from

an iOS device to aid a forensic analyst in drawing accurate conclusions. Although there are limitations to the information we are able to extract, which is discussed later in chapter 5, we can see that much of the success of this type of attack would depend on the user being unable to identify a malicious application running on his or her device. Continuously obtaining information about the running processes on the device, however, would allow an analyst to identify a malicious process belonging to a malicious application or an unusual parent process to a particular application which could lead to the conclusion that a malicious application has been running on the device.

Collecting information about the currently running processes on the device could also prove useful in defending against attacks by malicious applications downloaded from the official application store. In [7] the authors describe how it is possible for malicious applications to gain approval to be placed on the official iOS application store. This can be done by dynamically loading private APIs and then obfuscating the instructions in the source code that do the dynamic loading. Private APIs can allow for certain malicious activities to be carried out, such as programatically sending SMS messages from the device, and obfuscating the instructions responsible for loading those APIs allows the application to hide its malicious intentions from static analysis tools. These malicious applications, however, would likely not be able to hide their malicious intentions from an analyst who is able to view a history of the device's currently running processes. In regards to the example given in the paper by Han et. al., a malicious application that sends premium rate SMS messages using the device's SMS service could easily be detected over time since the "mobileSMS" process would be running whenever a message is sent. It would seem then that continuously collecting information about the currently running processes on a device is highly beneficial since it is useful in detecting mobile malware applications which are capable

of infecting a device different ways.

Another attack discussed in [8] by Tielei et al. shows how through the use of return-oriented programming (ROP), applications that are seemingly benign can pass the vetting process for the iOS Appstore only later to dynamically load private APIs triggered by some form of remote signal. Essentially any iOS app that attempts to load private APIs for cross-application communication will be quickly rejected by the AppStore once they are submitted, but by using ROP and control-flow hijacking, an attacker can bypass this measure and still use private API's and carry out malicious operations on the device such as sending emails, posting tweets, and sending SMS messages without the user's knowledge. Like with the previous attack, in order to improve the mobile forensic process, we need to consider what information we should continuously try to extract from an iOS device for an analyst to be able to identify this type of attack. Again because the attack revolves around placing a malicious payload onto a victim device, gathering process information over time would be useful in identifying potentially malicious applications, since the process used to launch the malicious application would only appear when the attack is being carried out. Additionally, because the end goal of this type of malicious application mentioned in the paper is to perform malicious operations, an analyst may also be able to identify those operations through the listed process IDs obtained from the extracted information and conclude that this type of attack is being carried out. Extracting network information could also be of use since it seems that the Jekyll application relies on placing vulnerabilities in the source code that can be exploited remotely. If an analyst is able to see both the running processes and any IP addresses that are actively trying to communicate with the device during the time of the attack, then he or she could potentially deduce that the attack is being carried out by correlating suspicious process information with suspicious network information.

In addition to platform specific attacks, to make our approach as robust as possible, we need to also consider attacks that are generic to all devices, such as network-based attacks, and continuously collect pertinent information accordingly. In [9] Cassola et al. discuss an "Evil Twin" network attack which can impact any type of wireless computer regardless of operating system. In the paper the authors describe that the "Evil Twin" attack consists of setting up a rogue wireless network that has an indistinguishable SSID from a valid wireless network. They consider a system where the valid network uses a valid authentication server and a valid certification authority to ensure normal users that the network that they are attempting to connect to is trustworthy. In the attack, the attacker establishes a rogue access point, a rogue authentication server and a rogue certification authority to mimic the valid system without raising any suspicion from the user. The attack proceeds by using what the authors call "reactive jamming," making it so that the wireless signal from the valid network is not detected by the victim user. Because the rogue network has the same SSID as the valid network as well as a certificate signed by a certificate authority, the user unwittingly connects to the rogue network at which point the attacker can obtain a hash of the user's credentials for that network. The hash is then brute-forced to reveal the user's credentials for the valid network, allowing the attacker access to the valid network with the same privileges as the user. This attack effectively demonstrates how any wireless-enabled device can be susceptible to an attack involving a rogue network, and although the goal of the attack mentioned in the paper was to steal the user's credentials, the user is prone to additional devastation by simply connecting to a malicious network. Naturally in order to identify this type of threat, a forensic analyst would need to have a record of any wireless networks that the mobile device in question had been connected to, so continuously collecting current network information such as an access point's SSID and BSSID would likely

prove useful in a forensic analysis.

After exploring several known attacks on mobile devices, and more specifically iOS devices, it seems that there are a few different types of information that should be monitored and collected from the device over time including but not limited to: currently running processes, current network traffic information, and current network access point information. To reiterate, the reason why continuously collecting information from the device over time is useful is because it gives a more complete set of evidence which then leads to more accurate conclusions about the state of the device. Bianchi et al. use this same reasoning in [10]. In the paper the authors talk about how when it comes to rootkits infecting a host machine, there is always some sort of evidence which can lead to the discovery of their existence. To ensure the discovery of rootkits, the tool described in the paper performs several different types of analysis including configuration comparison, code comparison, data comparison, and kernel entry comparison, and because their approach considers a variety of ways that a rootkit may attempt to hide its existence, and consequently a variety of different types of information, an investigator can draw accurate conclusions regarding the existence of rootkits when using their tool. Although this type of data collection is fairly new to mobile forensics, it is not completely unique to our approach. As previously mentioned, Scholnick draws attention to the importance of an on-device monitoring application and how it can potentially enhance forensics in [3]. He states "The core of any device's enhanced forensic potential, regardless of OS, will center on the availability of more comprehensive monitoring and tracking capabilities both inside the device and at the back end."

Other parties have made advancements in developing tools that continuously monitor and collect information from mobile devices, operating in a seemingly similar matter to the tool we developed in our approach. The most notable example of this

is AccessData's mobile endpoint monitoring capabilities that have been integrated into their ResolutionOne platform. According to [11], AccessData's implementation "detects unknown threats by providing visibility into network communications and running processes, so anomalous activities can be identified and remediated." At the time of writing this paper, however, no documentation is available regarding how AccessData's monitoring application actually operates, and no explanation is given regarding how a forensic analyst can actually identify the anomalous activities from the collected data. Other works have shown implementations for monitoring tools on the Android platform, such as the one discussed in the paper by Grover. In the paper, Grover talks about the implementation of a tool capable of collecting various types of evidences over time that can be used in a forensic investigation. Although the paper's discussion on monitoring screen lock status begins to show the tool's effectiveness in a forensic investigation [12], the tool does not explore methods that an investigator might use to improve their analysis given the information from the monitoring tool, whereas our approach does. In addition to offering a detailed explanation of how such a monitoring tool operates, we also show a proof-of-concept for how precisely an investigator could use the obtained information to form a more accurate conclusion of the events that transpire, thus enhancing the forensic analysis.

Chapter 3

PROPOSED APPROACH

After having discussed both why mobile forensics is important from a security perspective and how our approach seeks to enhance it, we now discuss our implementation process for our forensics tool. As mentioned in the previous section, our approach involves creating an on-device application to continuously gather current running process information, current network traffic information and current network access point information. That information is stored in logs on the device which are then sent to the server, where a Python script is run to parse the logs. After the server-side script parses the logs, the user is able to enter parameters to specify which sections of the collected information to analyze and how any anomalous information should be highlighted. This chapter is divided into four sections reflecting the steps in our approach: the Preliminary Investigation sections describes the background work done before we began designing our forensics tool; the System Architecture section describes the various components of our system, their interactions, and their purposes; the On-Device Application section describes the iOS application we created and some of the technologies used there-in to accomplish our goals; and the Server-Side Analysis section describes the analysis program we created as well as how the information is handled after it is sent from an iOS device to our server.

3.1 Preliminary Investigation

Before we began development of our on-device application, we were approached by a company that had had some of their sensitive information stolen from their mobile devices while their employees were on a business trip. They tasked us with

investigating one of their devices to try to find evidence of a compromise. The device that we were given to investigate was a 4th generation Apple iPad with Retina Display. In addition to this device we were also given an Apple iPad 2 which had no alterations to any of its settings and was supposed to represent a base model to which we were to compare the 4th generation Apple iPad. We began the investigation by taking logical images of both the device suspected to have been compromised and the base model device using AccessData's Mobile Phone Examiner Plus (MPEPlus) software. We then browsed through the logical image of the suspected device, seeing what types of information could be gleaned from a static image. Once we had an idea of what information MPEPlus was able to offer, we determined which of those types of information would be useful in figuring out whether or not the device had been compromised. Looking through all the relevant data extracted from the logical image, we attempted to find any evidence of malicious activity and tried to form a story of what malicious event, if any, might have occurred.

One of the pieces of evidence that we collected during our investigation of the suspected device was the times at which various applications were installed and uninstalled. This information was contained in the "MobileInstallation directory," which is under the "Logs" directory in the iOS file system and can be seen in Table 3.1. The reason why we decided that application installation information was relevant to the investigation was because if a malicious application were installed on the device, then the time stamp of the installation could potentially be correlated with other activity. Additionally, if we saw any anomalies in how long a particular application was installed, then we would have more reason to be suspicious of that application. For example, if an application was installed and then uninstalled a few moments later, then there would be a possibility that it was installed for a few moments only to carry out malicious instructions. Table 1 shows that a few of the applications were only

Application	Installation Time	Uninstallation Time
com.fmi.afaria	Sep 4th, 08:30:17	Sep 4th, 09:04:18
com.verisgin.mvip.iphone	Sep 4th, 14:51:17	Sep 4th, 14:53:24
com.sabre.tripcase.prod	Sep 7th, 15:17:11	Sep 18th, 23:08:25
com.hbo.hbogo	Sep 18th, 03:15:20	Sep 18th, 05:00:40
	Oct 8th, 04:37:00	Oct 14th, 11:27:19
com.amazon.Lassen	Sep 5th, 07:43:18	Oct 14th, 11:27:24
com.skype.SkypeForiPad	Sep 4th, 07:43:18	Oct 14th, 11:27:26
com.fishdog.hearts2	Sep 6th, 14:51:59	Oct 14th, 11:27:32
com.internationalsos.membership.isos	Sep 4th, 15:12:48	Oct 14th, 11:27:34
com.optimesoftware.Backgammon.free	Sep 6th, 15:13:01	Oct 14th, 11:27:36
com.pandora	Sep 5th, 17:29:58	Oct 14th, 11:27:38

Table 3.1: Mobile Installation Log

installed for a few hours or less, but once we researched the them a bit further, we found that they were not malicious.

Another piece of evidence we looked at in our preliminary investigation were device security and user settings, which determined a lot of the allowed actions on the device, located in the "PublicInfo" directory which is in the "configurationProfiles" directory in the iOS file system. The rationale behind looking into the user and security settings was that a malicious application could potentially try to disable specific settings to allow a malicious instruction set to be carried out, and by comparing the values for the user and security settings on the suspected device with those on the base-model device, we could see which settings were different and potentially maliciously altered, assuming the user did not tamper with the settings.

One of the more lucrative types of information that we investigated, in terms of what we were able to deduce about the integrity of the device, was the WiFi network information. We obtained the WiFi information from a plist file located in the "SystemConfiguration" directory which is in the "Preferences" directory in the iOS file system, and from this we were able to extract details about the various networks that the suspected device had connected to including the network's SSID, the BSSID, the channels over which the network was broadcasting, the country code, the time the device last joined the network, and the time the device last auto-joined the network. Additionally, elsewhere in the logical image file of the suspected device we were able to obtain the name of the vendors for each of the wireless routers in the plist file. Using all of this information, we were able to construct a rough timeline of networks that the suspected iPad had connected to, but what was most interesting about the WiFi information was that it revealed a potential rogue access point attack.

Three of the networks that appeared in the WiFi logs had similar SSIDs; they were "Garden Hotel", "GardeHotel" without a space character, and "Garden Hotel Free", respectively. The access point with the SSID "GardenHotel" and the one with the SSID "Garden Hotel Free" both were broadcasting on the same channels, had the same country code, and had the same device vendor. The access point with the SSID "Garden Hotel" did not share this information with the other networks with similar SSIDs. Additionally, the "Garden Hotel" access point did not have any value for a time when the device had last auto-joined that network, meaning that the access point may only have been available for a short time. This information, seen in Table 3.2, fits a scenario in which an attacker had set up a rogue access point with the SSID "Garden Hotel" in an attempt to steal sensitive information from any unsuspecting user who had connected to it. In this scenario it would make sense that the broadcast channel, country code, and device vendor would all be different for the rogue access

-	Garden Hotel	GardenHotel	Garden Hotel Free
Security Mode	None	None	None
Channels	11	1, 11, 6	1, 11, 6
Last Joined	2013-09-10 00:27:33	2013-09-10 00:27:46	2013-09-10 08:27:43
Last AutoJoined	-	2013-09-10 23:54:20	2013-09-10 23:55:16
Country	-	CN	CN
BSSID	00:1e:2a:07:7e:ce	38:22:d6:92:17:21	58:66:ba:ee:cf:01
Device Vendor	Netgear Inc	Hangzhou H3C T...	Hangzhou H3C T...

Table 3.2: Suspicious Network Nodes

point since the accuracy of those details is irrelevant for carrying out an attack where the goal is to trick users into thinking they are connecting to a valid access point.

Although we were able to extract various types of information from the device, we were never able to make any definitive conclusions about whether or not the device was compromised, only speculations. The installation and uninstallation times for all the applications allowed us to see whether or not an application was immediately removed after being installed, but even if that was the case, that information alone does not guarantee that the application was suspicious. Similarly, by looking at the user settings that we extracted we would have been able to tell if the device became less secure when compared to the base-model device due to changes in the user and security settings, but we would not be able definitively conclude that it was a malware application or some sort of malicious activity that caused the settings changes. Even with the WiFi network information, although we were able to conclude that there is a fair possibility that the anomalous network we observed could be the result of a rogue access point, we do not have the evidence to ensure that claim. It seems

that when it comes to obtaining evidence from a static image of a mobile device, there is a lack of confidence in any claim that an investigator can make regarding the integrity of the device due to the unknown surrounding circumstances to any single piece of information extracted. The solution to this problem then is to diminish the unknown elements of the surrounding circumstances so that an investigator can make conclusions about the device with a degree of confidence, i.e. there is a low probability that the conclusions are incorrect. Our approach seeks to accomplish exactly this by collecting various types of information from a device and correlating them with each other over time.

3.2 System Architecture

As previously discussed, there are two major components to our approach, those being an on-device application and also an analysis tool on a separate server. The purpose of the on-device application is of course to collect various types of information from the device over time using different methods and then to send that information to the server in the form of logs to be analyzed. The server then runs a script which parses through the information received and separates it into in different text files. The analysis program, which is stored on the same machine running the server, can then be run to analyze the text files created by the server script using different parameters specified by the user. These parameters indicate to the analysis program which additional calculations to perform as well as which information in the text files to perform the calculations on. This process can be seen in Figure 3.1.

3.3 On-Device Application

Because our preliminary investigation dealt with potentially compromised iOS devices and because we already had an understanding of what information was avail-

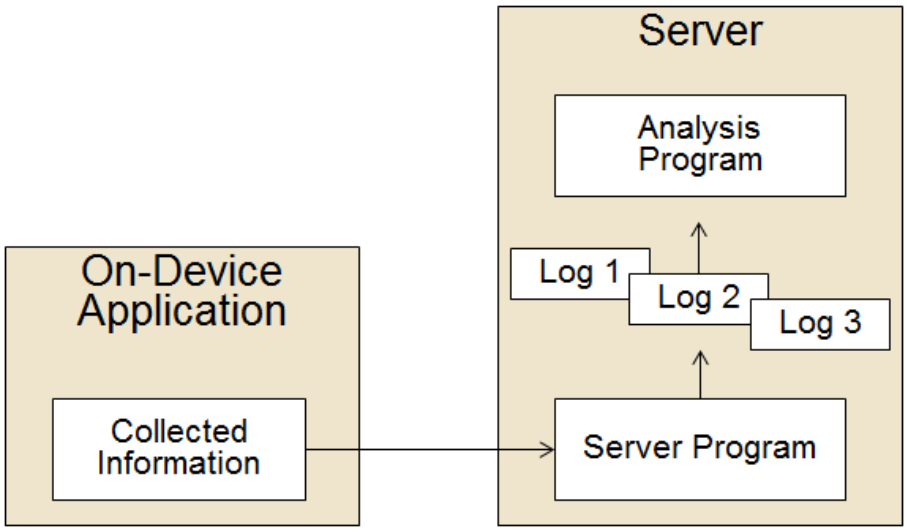

Figure 3.1: System Architecture

able to us, we decided to make our on-device application, which would continually extract information about the state of the device, for the iOS platform. The integrated development environment (IDE) we used to build and test our application was Xcode version 5.1.1, and the following Frameworks were imported into our project: XCTest.framework, SystemConfiguration.framework, AddressBookUI.framework, AddressBook.framework, MessageUI.framework, CoreGraphics.framework, UIKit.framework, and Foundation.framework. The device we used to test our application was an iPad Mini (1st Generation), and the machine on which we developed the application was a Mac Pro (Early 2008) running OS X version 10.9.3.

Regarding its design, our application was only ever intended to run in the background on the device and automatically collect and send information without any interaction from the user, so we did not develop a user interface for it; the only interaction that the user would need to have would be to simply launch the application like any other iOS application. The application itself is centered around a polling mechanism with a specific time interval, meaning that our application will

gather all the information it is designed to gather every five seconds until the application is closed. Creating background applications, however, comes with its own set of challenges. According to Apple's documentation about applications running in the background, only applications that are performing specific tasks are allowed to run in the background of a device without being suspended, one of which is an application that needs to update the user on the device's location [13]; thus in order to ensure that our application never entered a suspended sate, and therefore never stopped gathering information, we gave it instructions to check the device's GPS location during each poll every five seconds. This way the application had permission to run in the background and continue performing its functionality.

As previously mentioned, one of the primary sources of information that we want to continuously collect using the on-device application is the device's currently running processes. Obtaining this information lets an analyst see when, if ever, any suspicious processes are being run on the device. We accomplished this by creating a method in our code to extract the running processes, and we called that method whenever it was time for the application to perform a new poll, which was every five seconds. This method works by calling the "sysctl()" function from the "sysctl.h" C library which is used to effectively create an array of all the running processes at that time. We then create a dictionary object containing all the attributes of every process we just obtained. Using this method, we are able to collect relevant attributes for each process, such as the process start time, the process ID, and the process name. There are also additional process attributes available to us, such as process priority and process sleep time, but because our goal is to create a time-line of processes that are active at the time of each poll, we only need attributes that can help us identify a particular process, such as the first few attributes we mentioned. Once we have all the running processes for that particular poll, we place them in a string to be sent to

our server.

Using the on-device application to obtain the network traffic information allows an analyst to see when, if ever, any sort of suspicious or anomalous network traffic is occurring. Similar to how we collected the currently running processes, we collect the current network traffic by creating a network traffic collection method and calling that method within the poll method, which is called every five seconds. Our network traffic collection method is a modified version of the primary function in the "inet.c" file, which is an open source C file used by Apple in their implementation of the netstat program. We were able to modify the code so that instead of printing out the resulting network information to standard output, it saves the results to a string which is then passed back in to the main program. Our modified netstat function also only collects information pertinent to identifying and distinguishing specific network requests, such as the protocol used, the local IP address and port, the foreign IP address and port, and the state of the connection.

Our on-device application also continuously collects the current network access point information to gain a perspective on when, if ever, the device had been connected to an insecure or malicious network. A good example of the relevance of this information can be seen in our preliminary investigation where the WiFi logs revealed a potentially malicious access point. Again, in order to collect this information, we created a method that when called would obtain details about the current network node being accessed, and we then called this method in our poll method. This method works by using some of the functionality of the CaptiveNetwork library available for iOS development. Specifically, we used a method in the library called "CNCopySupportInterfaces" which allowed us to get an array of all the supported network interfaces. We then used the method "CNCopyCurrentNetworkInfo" to obtain information about the current network interface from the list of supported interfaces.

These methods gave us the SSID and BSSID of the network that the device was currently connected to; however, by using an online IP detection API with the web address "http://ip-api.com/line/?", we were able to obtain the IP Address, the latitude coordinate, and the longitude coordinate as well. Once we had the information obtained from the IP detection API, we formatted it to be sent to our server.

Every time the poll method is called, all the information that was collected during that poll is sent to the server. The way this works is that for each different data collection method, once the data is collected, it is formatted in a certain way and added to a global string. If for a particular poll, the current process information, current network traffic information, and current network access point information were all collected, the global string would then be composed of three separate sections. Each section would contain a sequence number, which represents how many times the poll method had been previously called at that point, and the data of one of the collection methods, that being the current process data collection method, the current network traffic data collection method, or the current network access point data collection method, along with the time that the data collection method was called. The collected information is formatted this way so that our server application can easily parse through the data it receives and identify which information belongs to which type of collected data and which sequence number each section of data belongs to. A sample of how the logString contents are formatted can be seen in Figure 3.2. The collected information is sent to the server by creating an HTTP request object and setting the contents of that request object to be the contents of the logString encoded as a data object. We then specify the HTTP method to be a "POST" method and send the request to the server synchronously.

Although the on-device application can operate as described, it is not without its share of usage limitations. One example of a usage limitation with the application is

```
Sequence Number: 39
Message Type: Current Process Information
2014-07-21 01:45:53
Process ID Process Name Process Start Time
2113       MobileSafari 2014-07-21 05:13:29
...

Sequence Number: 39
Message Type: Current Network Information
2014-07-21 01:45:54
Proto  Recv-Q  Send-Q  Local_Address        Foreign_Address     (state)
tcp4   0       0       192.168.1.139.50186  192.168.1.132.http  ESTABLISHED
...

Sequence Number: 39
Message Type: Network Nodes
2014-07-21 01:45:54
BSSID             SSID          IPAddress       Latitude  Longitude  ConnectionTime
68:1c:a2:0:8c:d0  GoodInternet  PrivateAddress  Unknown   Unknown    2015-06-17_03:42:36
...
```

Figure 3.2: Sample logString contents

that sending the collected information to the server requires an internet connection, which is not always available when using a mobile device. The problem with this is that it is possible that an analyst would end up trying to analyze an incomplete set of data due to some of the collected information not being sent to the server. We were able to mitigate this issue somewhat by instructing the application to store the logs locally if no internet connection is available; however, in order to not overload the application memory space, we limited the application to only locally store up to one hundred polls of information. Another instance of a usage limitation is that the most recent information poll on the device may not necessarily be synchronized with the data on the server since the log files of collected information may already exist on the server when the application is launched. Once the server receives information from the on-device application it detects the information's sequence number and writes that poll information to a log file. The deatails of this process will be explained in the subsequent sections. Each log file on the server has ten instances of consecutive poll information, meaning there would be a problem with sending information to the

server if, for example, the application is launched and tries to send poll information with sequence number zero (the first poll of the application) to the server, but a log file already exists on the server that contained poll information with sequence number zero from a previous session. The result would be that the additional sequence zero information would be appended to that particular log file, which would cause our analysis program to crash since it expects log files with at most ten instances of polled information. The way we mitigate this is by telling the application to first retrieve the most recent log file from the server and determine what the most recent sequence number is in that log file. This way the current session will start where the previous one left off.

3.4 Flask Server

The data collected by our on-device application is first received by our server, which is responsible for organizing the information for analysis later. Our server uses the Flask web application framework, specifically version 0.10.1, for handling the various HTTP requests sent from our application, and because of this, we were able to implement all of our instructions on handling communications from the on-device application in a single python source file which we called "launcher.py". According to the Flask API documentation, once a Python file that uses the Flask web framework is run, a global Flask object is created which contains instructions for how to handle URL rules and also code for launching a local web server application, which is activated on launch. The Flask web framework also allows us to specify URL rules to call certain functions when receiving an HTTP request for a specific route on the server [14]. This then allows us to perform some actions upon receiving an HTTP request from the on-device application, which we use to organize all of our extracted data for analysis.

There are three primary routes used in the server's communications with the on-device application: "/hello", "/startup", and "/extraLogs". The "/hello" route is where the on-device application sends the requests containing the data of the collected information from the device. The on-device application sets the request method to "POST", so it does not expect a response from the server, and sends the request to the "/hello" route, which is just the base URL of the server with "/hello" appended. Upon receiving the request at the "/hello" route, the launcher.py program converts the data in the request to a string and writes it to a log file in the current directory. It accomplishes this by first using a regular expression to find the sequence number of the received data. It then appends the received data to a specific log file which it determines by dividing the corresponding sequence number by 10 and rounding down to the nearest whole number. The log file naming convention is the word "Output" followed by the number it calculated by dividing the sequence number by 10 and then the ".txt" extension. For example, if the sequence number of the received data was "39", it would write the received data to the file "Output3.txt". Using this method, every log file will have 10 consecutive instances of the data extracted from the mobile device with no limit to the number of "Output" log files possible.

The "/startup" route is responsible for letting the on-device application know which sequence number to start at when it is first launched. As previously mentioned, one of the usage limitations for our on-device application is that occasionally log files already exist on the server from a previous session, and in order to avoid writing collected information to log files that already contain 10 sets of collected data we need to determine the sequence number of the most recent data written to the log files on the server. The way we accomplish this is by having the on-device application create an HTTP request object, setting the request method to "GET", meaning that it will be expecting a response back from the server, and we send the request to the

"/startup" route on the server. Upon receiving the GET request from the application, the server creates a list of all the files in the current directory, that being the same directory where the "luancher.py" file is located, that have the ".txt" extension by using a regular expression. This effectively creates a list of all our current "Output" log files. We then use another regular expression to search through each file in our list of files and find all the sequence numbers in each file. We then take the highest sequence number we found and set that as our return value to send back to the on-device application. In this way, we tell the on-device application what the most recent sequence number that the server received was.

The "/extraLogs" route is used for receiving collected information that was temporarily stored locally on the device. As previously mentioned, another limitation of the on-device application is its need for an internet connection to save the collected information from the device. The way we handle this limitation is by having the application store the logs locally if there is no internet connection and then send that information to the server at a later time. This locally stored information cannot be sent to the server through the normal "/hello" route; however, since the locally stored information may consist of information collected over many sequences instead of just one sequence, which is what the "/hello" route would be expecting. The on-device application, therefore, sends the locally stored information to the "/extraLogs" route once an internet connection is detected. Once the server receives the request on the "/extraLogs" route it parses through the request data, which is a compilation of the locally stored instances of collected information, and uses a regular expression to make a list of all the line numbers from the request data that start a new sequence number. Once the list of line numbers is made, the server program then creates a list of sections of the input received from the on-device application based on the list of line numbers. In other words, we just split up the original input based on where a new se-

quence number starts, so we get a list of individual sequence instances, each one being similar to what the on-device application would normally send to the server through the "/hello" route. The server program then iterates though this list of sequence instances and adds them to an "Output" log file based on the sequence number, like it would do through the "/hello" route. In this way, data stored locally on the mobile device is not lost and can be stored in the log files on the server normally.

3.5 Analysis Program

After the data received from the on-device application is organized into individual log files by the server program, the data is ready to be processed by the analysis program. The primary goal of the analysis program is to provide a way for a forensic analyst to draw conclusions with a high level of confidence about the state of the device, and the specific features of our analysis program that aim to achieve this goal are what makes our approach unique when compared to the existing approaches. In the online article "Mobile Security for a Nomadic Workforce", the author Lee Reiber, who oversees all mobile forensics activities for the company AccessData, talks about the capabilities of the company's mobile endpoint monitoring software. He states, "It also detects unknown threats by providing visibility into network communications and running processes, so anomalous activities can be identified and remediated" [11]. Although this technology can continuously collect information from mobile devices to allow anomalous activities to be identified, it does not show how this information can be used to identify anomalous activity, meaning that it does not propose a way for an analyst to effectively use that continuously collected information.

3.5.1 Time-Line Table

Like our server program, our analysis program consists of a single python file named "logAnalyzer.py", which when launched will perform a various set actions depending on the arguments it receives. One of the primary features of this program is constructing a table that depicts the activity of each element of data that was collected from the on-device application over time. The first task that the program performs upon being launched is to scan through all the "Output" log files in the current directory and create a dictionary of all the sequences, or instances of polled data, from the device. We do this so that all of the collected data from the "Output" log files are easily accessible and traceable for when the program is constructing the time-line table structure. The way it does this is by first creating a list of all the "Output" log files, and then for each log file in that list, it creates a dictionary of sequences, appending all the key-value pairs to the dictionary of sequences of the previous log file. Then for each sequence in the sequence dictionary, the program creates another dictionary containing several lists, one for each type of data collected for that sequence. This is done by scanning through the log files line by line and using regular expressions to determine when a particular type of collected data is being scanned. The result is a comprehensive dictionary of all the collected data where each key is a sequence number and each value is another dictionary in which each key is a type of collected data and each value is a list of all the elements of that type of collected data. This structure is shown in Figure 3.3.

The next major task that the analysis tool performs is constructing the actual time-line table structure from the comprehensive dictionary of all of the collected data. The design of the time-line table is that all of the sequences from a set of collected data are represented as different columns in the table and each individual

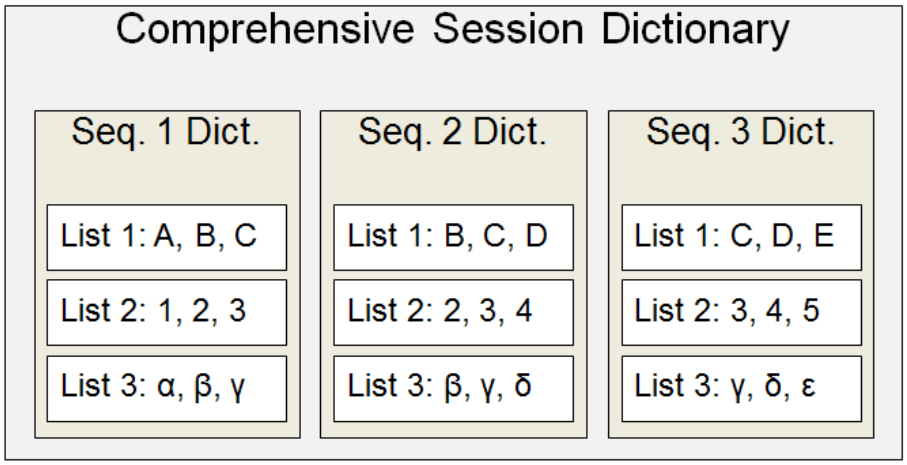

Figure 3.3: Comprehensive Structure Created from Output Logs

data element from each type of data is represented as its own row. This design allows an analyst to easily compare different data elements and different sequences to one another, allowing him or her to see if any data elements or sequences are suspiciously similar to one another or uncharacteristically different from one another. A challenge with this design, though, is that making a row for each data element from every sequence would mean that there would be a lot of repeated rows in the final time-line table structure, since a lot of data elements persist over several sequences. The solution to this is to systematically make a list of all the unique elements to use for creating the rows in the time-line table; however, because the data elements are further separated into different lists based on the type of data in our comprehensive dictionary, we can first create lists of unique data elements based on the different types of data, which would allow the analyst to filter the time-line data structure based on data type (discussed further in subsection 3.5.4). We can then append these lists of unique elements together to produce a single list of every unique element across all types of collected data for the session that the on-device application was running. This process can be seen in Figure 3.4.

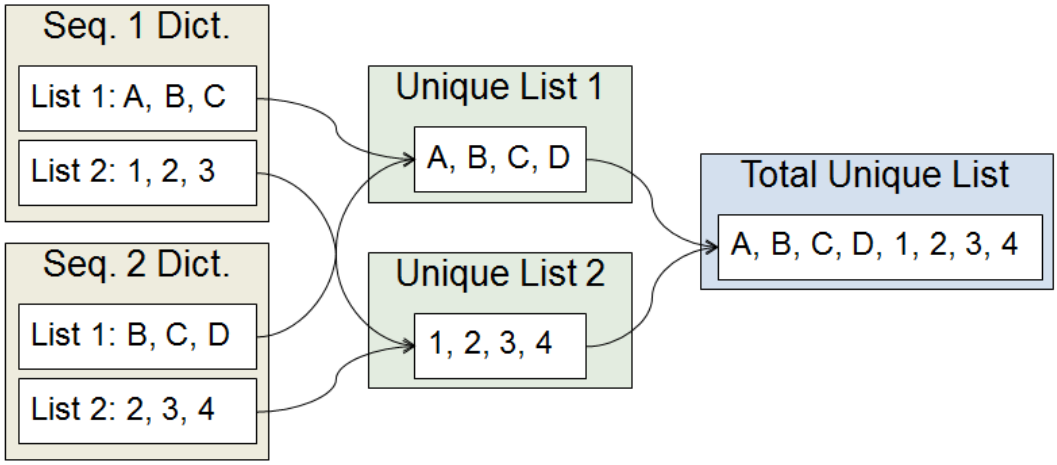

Figure 3.4: Creating Lists of Unique Elements from Monitoring Session

Once we have a list of all the unique data elements we can begin to fill in the cells in the table. Each cell that is not the first cell in a row or a column represents whether or the data element at the beginning of the row it is in is active during the sequence at the beginning of the column its in. To fill in the cells, we simply check to see which sequence and data element a particular cell correlates to and then check the comprehensive dictionary structure to see if that particular data element is present in the list of elements for that sequence. If it is, then we write a "1" in that cell representing that that data element was active; otherwise, we write a "0".

Regarding the actual table that the "logAnalyzer.py" program outputs, the first row of the time-line table is the header row, and it is simply the different sequences and their respective time-stamps taken from the Output log files. Once we have a list of all the unique data elements we can begin to fill in the cells in the table. Each cell that is not the first cell in a row or a column represents whether or not the data element at the beginning of the row its in is active during the sequence at the beginning of the column its in. To fill in the cells, we simply check to see which sequence and data element a particular cell correlates to and then check the comprehensive dictionary

structure to see if that particular data element is present in the list of elements for that sequence. If it is, then we write a "1" in that cell representing that that data element was active; otherwise, we write a "0". Fully copying each unique element as the first item in each row of our table, however, is not feasible since each element is too large in terms of character length to conveniently fit into a single cell of our time-line table. As a result we pick out the most identifying attributes of each type of data so that we can abbreviate what is actually being represented as the first item in each row of our table. For the process data type we use the process name; for the network traffic data type, we use the traffic's foreign address; and for the network access point data type, we use the BSSID. Abbreviating data like this causes a loss of information available to the analyst when looking at the time-line; however, because we include key identifiers for each element, the lost information can be easily looked-up in the "Output" log files. Additionally, an analyst is still able to use the time-line table to effectively draw conclusions, since abbreviating the information in the table has no impact on an analysts ability to relate different types of information to each other or identify certain elements of information as being anomalous. As an example, Figure 3.5 shows how obtained network traffic information found in the "Output" log files is abbreviated when placed in our Time-line Table.

As previously mentioned, the purpose of creating a system that allows for continuous collection of information from a mobile device is to provide an analyst with a more complete story of the events that occurred, allowing him or her to make conclusions with a higher level of confidence than those that could be made from a commercially available forensics tool alone. For example, say that the on-device monitoring application is able to collect 2 different types of information from the device, process information and network traffic information, over a period of sixty seconds. If a poll occurs every 5 seconds, then there would be 12 polls, which in our analysis program

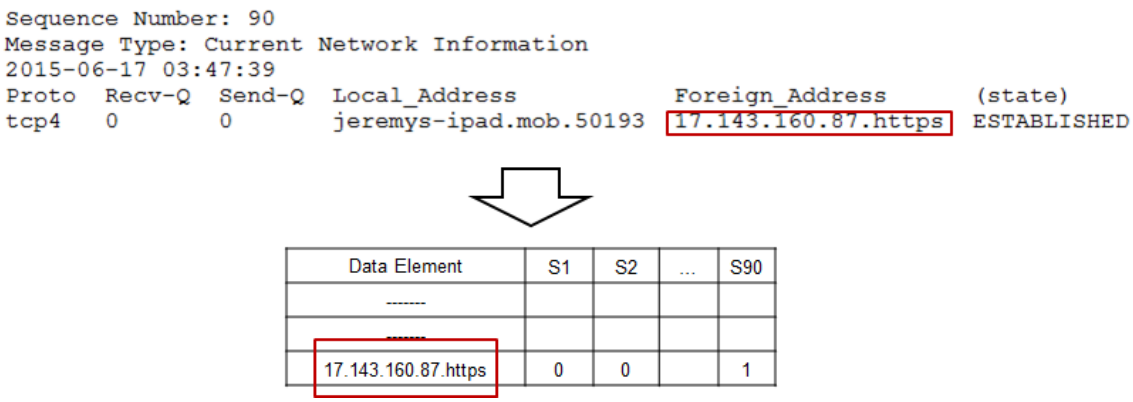

Figure 3.5: Abbreviating Data from the Log Files for the Time-Line Table

would be represented as sequences 0 through 11. Suppose that the analyst uses the analysis tool to produce the time-line table output similar to that seen in Table 3.3 and notices that Process C is active only at sequence 9 and that at sequences 10 and 11 there is network traffic between the device and a server located in a foreign country. The analyst would likely determine that there is a high probability that the process observed in sequence 9 is malicious since it is anomalous and occurs directly before the suspicious communications observed in the following sequences. In this case the analyst is essentially able to determine more unknown factors about the events that occurred on the device by looking at data collected over a period time as opposed to using a commercial forensics tool that looks at a static image of the device, thus allowing for a more complete analysis. In the case of a static image, if an analyst were to obtain an image of the device through a commercially available tool, he or she might know that the process in question was launched at some point and that the network communications to the foreign servers occurred, but it would be unknown how those events related to each other, leaving the analyst with little information to make a conclusion. Perhaps the analyst would be able to find log files showing start times of process and network activity on the static image of the device, but log files

Sequence	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11
Process A	0	0	1	1	1	0	1	1	1	1	1	1
Process B	1	1	0	1	0	0	1	1	0	0	0	0
Process C	0	0	0	0	0	0	0	0	0	1	0	0
Local Address	1	1	0	0	1	1	1	1	1	1	1	1
Foreign Address	0	0	0	0	0	0	0	0	0	0	1	1

Table 3.3: Sample Time-line Table

typically do not give any insight as to the length of time that an event persists as seen in the WiFi access point logs in our preliminary investigation.

Naturally, the works related to our thesis also propose mechanisms for continuously collecting data in order for analysts to make conclusions with greater confidence, but by creating a time-line table, our approach provides a way for the analyst to use that collected information effectively, unlike the related works. In the hypothetical situation given, an analyst is able to use our time-line table structure to quickly identify that a process is anomalous, since only one sequence in the table shows that process, and also that there is a high probability that the process is related to the suspicious network activity, since the process is active only directly before the suspicious network activity. Essentially, our analysis program organizes the continuously collected information by using our time-line table structure to improve the forensic process.

3.5.2 Levenshtein Distance Calculation

Although our time-line data structure is a good example of a way to use the continuously collected information to improve a forensic analysis, it is not the only

improvement that we can make. In addition to organizing the collected data, we can also perform various calculations on the data to highlight for the analyst which parts of the data should be investigated thoroughly, thus improving the efficiency of a forensic analysis. One way to do this is by calculating the Levenshtein distance for each row in the time-line table. Levensthein distance describes the number of changes that need to be made between two sequences of elements in order to make them the same, and it has been shown to be useful in the field of computer forensics by reducing the data to be searched through. Mangnes discusses this technique in [15]. In the paper, Mangnes mentions how when searching for a specific piece of data within a forensically obtained set of data, the data can be split up into a number of data buffers and then searched through using a Levenshtein distance calculation. The calculation would compare the piece of data that is of interest and any one of the data buffers and return a Levenshtein distance value. The data buffer that corresponds with the lowest Levenshtein distance value would be most likely to contain the piece of data being searched for and then a more precise search can be performed on that specific data buffer to find its exact location.

Similar to the approach in the paper by Mangnes, our approach uses Levenshtein distance calculations to find similarities between sets of data. The difference, though, is that instead of using Levenshtein distance to search for specific data, our approach uses the calculation to find correlations among different pieces of evidence, which if investigated further, may allow an analyst to quickly arrive at a conclusion about the state of the mobile device. For example, If two rows in our table have a very small Levenshtein distance over a large number of poll sequences, then it is very likely that those two rows are related to each other, and if the analyst determines that the evidence represented in those rows are typically not related to each other, then that evidence potentially represents malicious activity. In this way the Levenshtein

distance can highlight an area that should be further investigated.

For example, assume an analyst has a set of monitored data collected from a mobile device which shows several processes and several instances of current network traffic across 50 poll sequences. The analyst calculates the Levenshtein distance for each data element, comparing each element to every other element in the set. Once the calculation is complete, if the analyst notices that two seemingly unrelated data elements, an unrecognized process and a network traffic instance showing a connection to a foreign address, have a Levenshtein distance of 0, then the analyst would determine that there is a suspicious relationship between those data and likely investigate them further. In short, low Levenshtein distance values between elements in a data set often indicate some sort of relationship, which can occasionally be malicious if the analyst can determine that the elements are not typically related.

Our "logAnalyzer.py" program includes functionality to calculate the Levenshtein distances for our data. For the calculation, we use the "Levenshtein" Python library which contains a comparison function that can calculate the Levenshtein distance between two strings. To perform the calculation we essentially format each row in our time-line table into a single string object of 1's and 0's representing when the element for that row was active or inactive. We then place all of these strings in a list and loop through that list, performing the Levenshtein distance comparison on every other element in the list for each element in the list. The results are then stored in a table with both the columns and the rows being the list of all the elements in our extracted data set. This process is illustrated in Figure 3.6; note that in the Resulting Data table the values on the diagonal are not included since any element string will have a Levenshtein distance of "0" when compared to itself.

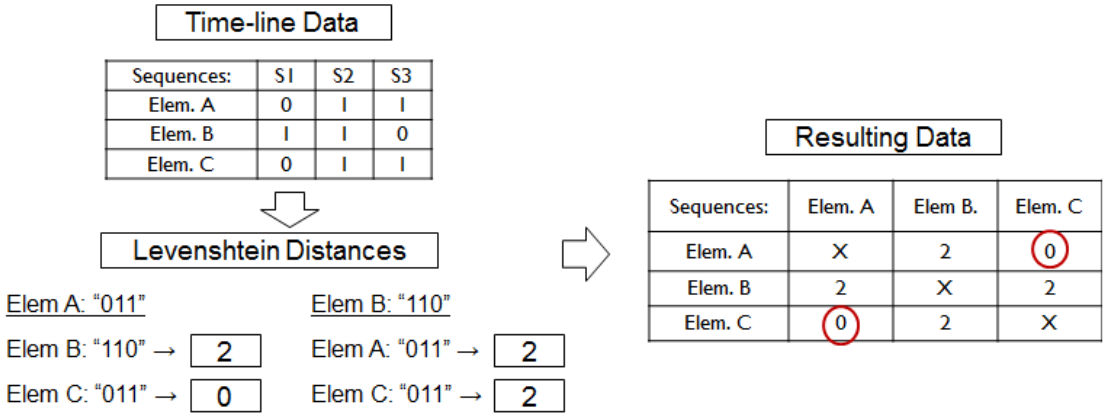

Figure 3.6: Levenshtein Distance Calculation

3.5.3 Euclidean Distance Nearest Neighbor Calculation

The other calculation that our "LogAnalyzer.py" program is capable of performing on our data to highlight areas of interest for an analyst is a Euclidean distance calculation. Euclidean distance is the mathematical term for the distance between two points in Euclidean space, and is the metric used in calculating the *k-th* nearest neighbor as decribed in the paper by Ramaswamy et al [16]. In this paper, the authors use the Euclidean distance calculation to find any outliers in a set of data points. The way they do this is by first determining a threshold number of points, n, of the total points in the data set that would be considered outliers. Then by calculating the Euclidean distance from each point to every other point in the set, the authors determine the distance of the *k-th* nearest neighbor for each point in the set. If for a particular point there are no more than n - 1 other points that have a higher *k-th* nearest neighbor value, then that point is considered an outlier of the data set. In another paper by Lazarevic et al. the authors demonstrate how various outlier detection schemes can be used in anomaly detection. One such outlier detection scheme was a modification of the *k-th* nearest neighbor outlier detection

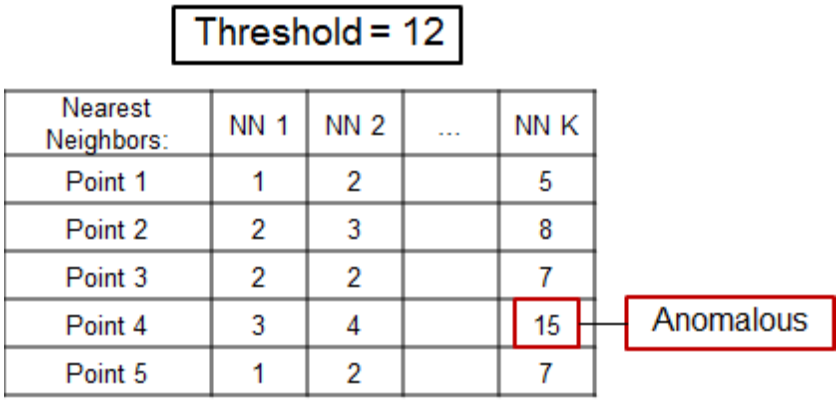

Figure 3.7: Nearest Neighbor Anomaly Detection

scheme described in the previously mentioned paper. The modification consists of replacing the previously described threshold number of points in a data set, used to determine the outlier points, with a numerical threshold value. Any point that has a *k-th* nearest neighbor distance greater than the threshold value is considered an outlier [17]. The authors tested the effectiveness of this outlier detection scheme, among other schemes, as an anomaly detection scheme by seeing what percentage of anomalies in the 1998 DARPA Intrusion Detection Evaluation Data set it was able to detect. According to the resulting data, this particular anomaly detection scheme was able to detect a higher percentage of anomalies associated with the network attacks than any other anomaly detection method described in the paper. Figure 3.7 shows how the nearest neighbor calculation can be used in anomaly detection.

An assumption one could make about malicious activity is that it tends to happen significantly less frequently than non-malicious activity. This assumption is supported in [18]. In the paper, the authors discuss how alerts from an intrusion detection system can be prioritized and grouped based off anomalous behavior. They carry out attacks on a test system and use their own tool to group intrusion detection alerts based off of how anomalous they are, giving higher priority to the alerts that are more anomalous.

When analyzing the higher priority alerts, they found that the true positive rate was 100 percent, meaning that no suspicious or malicious behavior was left out of the high-priority grouping of the alerts, which implies that anomalousness is in fact a property of malicious activity. Because of this property we decided to use Euclidean distance *k-th* nearest neighbor calculations to find anomalous, and therefore suspicious, data in our data sets, similar to how the modified *k-th* nearest neighbor outlier detection scheme was used in the paper by Lazarevic et al. In order to calculate the nearest neighbor Euclidean distances on a data set we receive from the on-device application, we first need to specify what defines a data point within our data. The data points that we use in our Euclidean distance calculations are essentially the different columns from our time-line table structure where each value in a column, either a "1" or "0", represents a value for a different dimension in Euclidean space. This means that if, for instance, there were a data set consisting of m data elements across n poll sequences and we want to find the k nearest neighbors for each point, then we would be performing Euclidean distance calculations between sets of 2 of n points across m dimensions and keeping the lowest k distances for each of the n points . Once we have our k nearest neighbors for each point, we create a new table to host those results, in which there are $k + 1$ columns, which list the k nearest neighbors for each of the $n + 1$ rows, which represent each of the poll sequences. This resulting Euclidean distances formed from a data set represented in a sample time-line table can be seen in Figure 3.8.

Naturally a concern that arises from this type of calculation is that if we have an arbitrarily large number of dimensions across which we are trying to calculate Euclidean distances for every point in our data set, then the calculations may become computationally impossible - also known as the curse of dimensionality. To solve this problem we employed the use of a ball-tree data structure to organize our data points

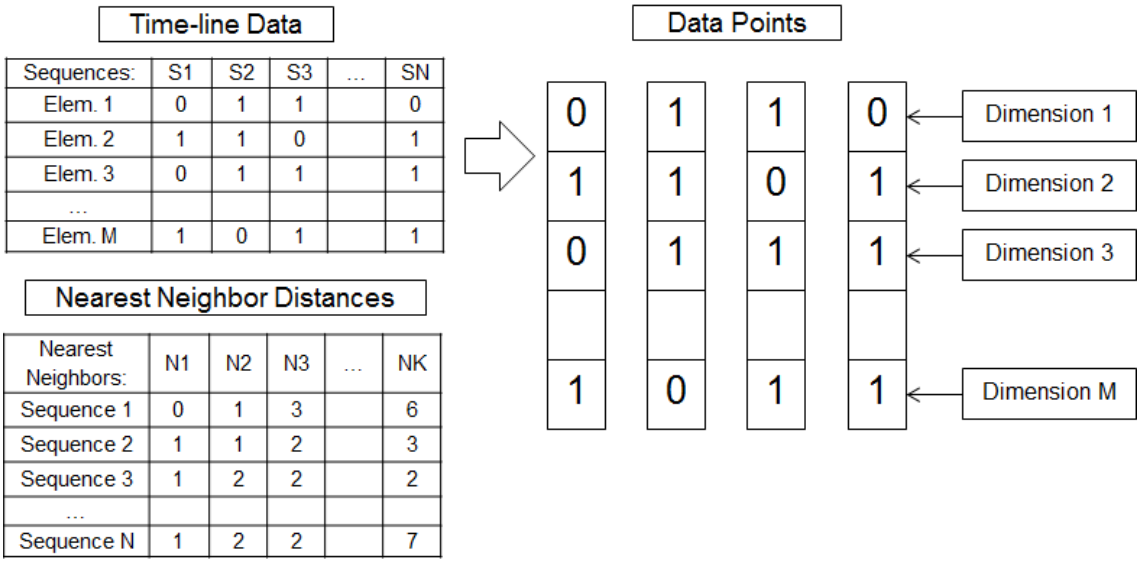

Figure 3.8: K Nearest Neighbor Distance Calculation

and calculate the k-nearest neighbors for each point in a computationally feasible amount of time. The ball-tree data structure is a type of binary tree where each node defines a hypersphere with half of the dimensions of its parent's hypershphere - its sibling node defines a hypersphere with the other half of its parent's hypersphere's dimensions. Each node also contains all of the points whose distance is closer to its centroid than the centroid of its siblings hypersphere [19]. A ball-tree data structure, therefore, has a special property in that for any point, p, in a ball tree, the distance between p and another point in the tree will always be greater than or equal to the distance between p and the centroid of any hypersphere containing that point [20]. What this means is that when searching for the point in our set of data closest to a point, p, any subtree and all the points therein that are further away from the closest point we have observed so far, which is easily discernable due to the organization of the data structure, can be ignored for the rest of the search. Continuing the search, we can keep track of whatever point is currently closest to point, p, and

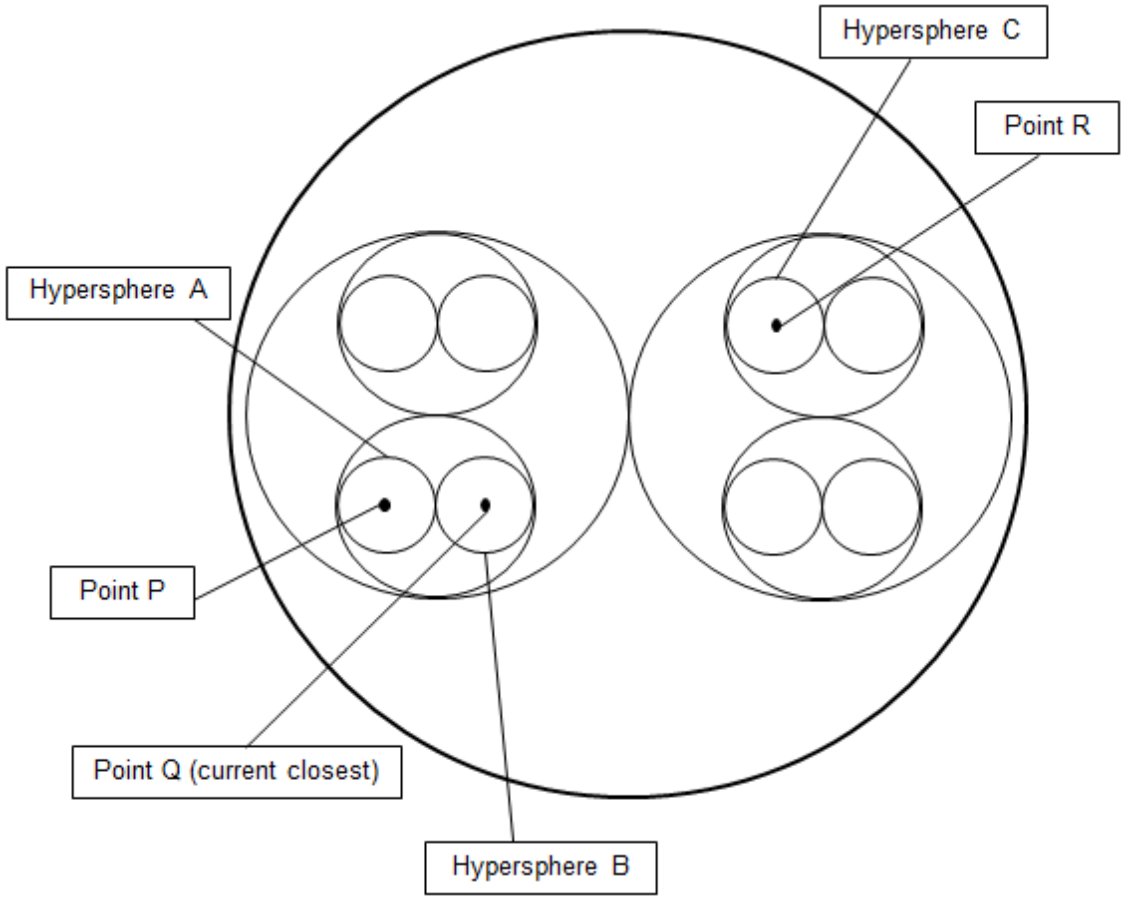

Figure 3.9: Nearest Neighbor Calculation Using Ball-Tree Data Structure

whenever a new closest point is found, we can discard the old closest point as well as any points within subtrees further away from the new closest point. Eventually we will discover the nearest neighbor to point p without having to have calculated the Euclidean distance from p to every other point across an arbitrarily large number of dimensions. This process is illustrated in Figure 3.9

Figure 3.9 shows a snapshot of a search for the nearest neighbor to point P in a data set. In the figure we can see that point P shares a parent hypersphere with the current closest point, point Q. When attempting to calculate if another point, for example Point R, is closer to point P than point Q, we can simply check where

the hyperspheres that contain the points are located in the ball-tree data structure. Because it takes more hops to get from hypersphere A to hypersphere C than it does from hypersphere A to hypersphere B, we know that point Q is closer to point P than point R is. The computational complexity arises, however, when trying to calculate which hyperspheres contain point R since there are an arbitrary number of dimensions. Fortunately, because we only care about whether or not point R is closer to point P than point Q, we can stop trying to locate point R's location in the tree as soon as we see that one of its ancestor's hyperspheres is farther away to hypersphere A than hypersphere B is, saving a considerable amount of computation.

To demonstrate how discovering anomalies by calculating the k nearest neighbors from a data set can aid in a forensic investigation, we consider a scenario in which a data set contains over 200 data elements across more than 500 poll sequences. In this data set there are records of the mobile device connecting to a malicious WiFi access point as well as several web addresses that were only communicated with while connected to the malicious network, but other than these events, the activity on the device is generally constant. This means that in an investigation an analyst could potentially determine, by looking at the time-line table, that the sequences describing connections to those specific web addresses while on that particular network are anomalous and therefore somewhat suspicious, but because the data set is somewhat extensive, it is unlikely that the analyst would be able to detect the anomalous sequences. Using Euclidean distance calculations, however, an analyst can gain an understanding of how different each sequence is from the other sequences, and because the sequences that show the connections to the specific web addresses while connected to a malicious network are uncommon among the entire set of sequences, they would show an abnormally high *k-th* nearest neighbor value. For example, if there are four sequences that show the connections to specific web addresses while

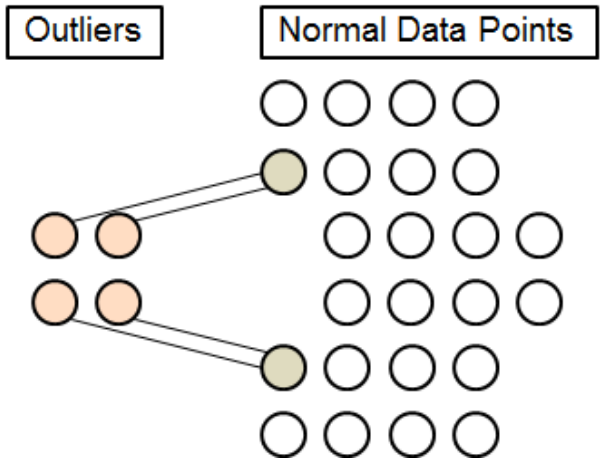

Figure 3.10: K Nearest Neighbor Euclidean Distance Anomaly Example

connected to the malicious network, then the fourth nearest neighbor distance value for those sequences would be abnormally high when compared to the fourth nearest neighbor distances of all the other sequences. In this way an analyst could determine more easily that those sequences are anomalous which could lead to him or her discovering the connection to the malicious network. This scenario is illustrated in figure 3.10. In the figure, the outlier points, which represent the four sequences in which a connection to the malicious network was made, clearly are anomalous when compared to the normal data points, but if we were to only look at the first three nearest neighbors for each point, then they would not appear anomalous or suspicious. The lines connecting the outlier points to the normal data points, however, represent the fourth nearest neighbor distances for the outlier points, which are much more evidently anomalous compared to the fourth nearest neighbor distances of any other point.

Although the time-line table structure, Levenshtein distance calculations, and Euclidean distance calculations in our approach are all effective ways in which the continuously collected information can be used to aid an analyst in an investigation,

calculating the Euclidean distance to find the k nearest neighbors for each sequence is somewhat unique in that it helps to identify how anomalous something is, which is a property often associated with malicious activity, as opposed to calculating the Levenshtein distance which only points out correlations among the data elements and relies on the analyst's intuition to draw meaningful conclusions. As previously mentioned, works related to our approach have drawn attention to the idea that by continuously collecting information from a mobile device, it is possible to perform some form of anomaly detection which would help an analyst identify malicious activity, but those works never actually proposed how this can be accomplished. This feature of our approach, therefore, offers meaningful contribution to the area of mobile forensics, since we are specifically exploring effective ways in which the continuously collected information can be used to improve a forensic analysis.

3.5.4 Additional Features

Another one of the major features of our "logAnalyzer.py" program is that it can take in several command-line arguments that allow the user to manipulate the different outputs of the program in various ways. For the time-line table, there are two parameters "-S (–sequences)" and "-R (–rows)" that allow the user to specify which sequences and rows are displayed in the time-line table, respectively. This means that if an analyst has already determined that a specific set of sequences and rows to be of interest in an investigation, he or she can specifically view that data in order to more efficiently continue on with the investigation and draw conclusions. Additionally, the user is able to filter the data that gets displayed in the time-line table by the data type. There are three parameters "-P (–processes)", "-A (–addresses)", and "-N (–networks)" that allows the user to only view specific types of data in the resulting time-line table, and similar to the row and columns filtering parameters,

these parameters are useful in that they can allow an analyst to more efficiently arrive at a conclusion by limiting the data to be analyzed.

The Levenshtein distance calculation output table also has its own command-line input parameter which is "-L (–levenshtein)," but instead of filtering the data that is displayed in the table, the argument highlights certain cells within the Levenshtein output table. This argument takes in a numerical value which all levenshtein distances in the table are compared to. If any Levenshtein distance is less than or equal to the argument's value, then two asterisk characters are appended to the number in the table. This allows an analyst to specify the maximum Levenshtein distance value that would be of interest in the investigation and then easily see which entries in the Levenshtein distance output table are below that specified value. Essentially this parameter makes it easier for an analyst to identify data elements that have a strong correlation with one another, and are therefore of interest, which improves the efficiency of the analysis.

Like the other output tables, the k nearest neighbors output table also has a corresponding input argument which is "-E (–euclidean)", and it takes in a numerical value which specifies how many nearest neighbors will be calculated and displayed for the sequences in the output table. Naturally we need to allow an analyst to specify how many nearest neighbor distances should be calculated for the sequences so that if there are multiple anomalous sequences in a data set, like in the example previously described, the analyst will be able to determine how many anomalous sequences there are. Unlike the other input arguments, however, this argument does not improve the efficiency of the forensic analysis as it does not filter out unimportant data or highlight data of interest. Our program does, however, implement a highlighting mechanism specific to the k nearest neighbor calculation. In order to ensure that a forensic analyst is able to efficiently identify any anomalous sequences in the k nearest neighbor output

table, we calculate both the average and the standard deviation for each of the nearest neighbors 0 through k across every sequence. To calculate the standard deviation for any one of the nearest neighbors 0 through k, we use the formula:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1}}$$

In the formula σ is the resulting standard deviation, x_i is a single data value, $\bar{x}$ is the mean of all the data values, and n is the number of data points. For every sequence in the output table we compare the value of each nearest neighbor 0 through k to the average value of that nearest neighbor. If the nearest neighbor distance value for that sequence is greater than the average value for that nearest neighbor by at least the value of the standard deviation, we append two asterisk characters to the end of the value in that cell, specifying that for that sequence, that nearest neighbor distance value is outside of standard deviation and is therefore suspicious. Additionally because the k nearest neighbor distance metric is unique in that it can be used to identify a property that is often associated with malicious activity, as previously described, we also append an asterisk to every cell in the time-line table that belongs to a data element (a row) that is active at some point in the table during a sequence that was shown to have a k nearest neighbor distance value outside of standard deviation. Figure 3.11 uses an example set of data to illustrate this process further. Due to these modifications to the k nearest neighbor and time-line output tables, we are effectively highlighting any pertinent information in those tables which in turn can improve the efficiency of the forensic analysis.

After determining the nearest neighbor distance values that are outside of standard deviation, we can use a similar process to calculate anomalous data elements in the time-line table structure. We first get a list of all the sequences in the Euclidean distance nearest neighbors results table that contain at least one nearest neighbor

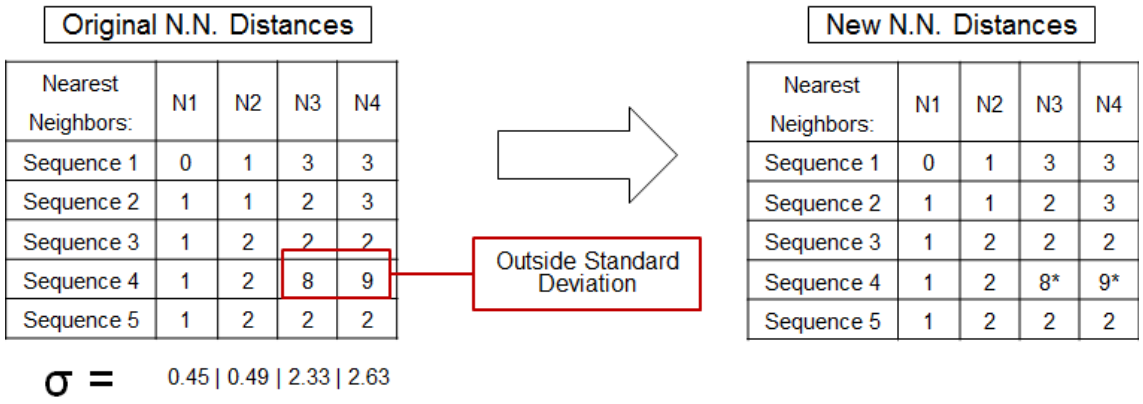

Figure 3.11: Highlighting Anomalous Nearest Neighbor Distances

distance value outside of standard deviation, in other words, a list of the sequences that have at least one value followed by asterisks. We again calculate standard deviation and average values, but this time we calculate those values for each row in the time-line table structure. If any call for any row of the time-line table structure contains a value that is outside of standard deviation and that cell also is within the column of a sequence that is in the list of anomalous sequences from the euclidean distance nearest neighbors results table, then that row in the time-line table structure containing that cell has two asterisks appended to each of its cells. In this way, we can highlight every data element in the time-line table structure that is related to the anomalous sequences, since during those sequences, those particular data elements also had abnormal values.

Chapter 4

EVALUATION

The goal of our approach is twofold: First, it is to demonstrate that by continuously collecting information, an analyst can perform a more accurate assessment of a mobile device, and second, we show how specific methods can be used to interpret the collected information to form meaningful conclusions. The first part of our goal is achieved by demonstrating the discrepancies between the information collected from a commercial forensics tool and the information we were able to gather using our on-device application. In regards to the second part of our goal, although we can explain our process for interpreting the collected data to form more meaningful conclusions, demonstrating the effectiveness of our approach using actual data gives it credibility and helps solidify our claims; therefore, we simulated a network-based attack on an iOS device similar to the "evil twin" attack previously mentioned in the Related Works section. Our simulated attack is similar in the sense that it involves creating a malicious access point with an SSID that resembles that of a non-malicious one. Whenever an iOS device connects to the malicious access point and tries to access a site hosted on a specific server, the request is forwarded to another malicious server which hosts a malicious site without the user knowing. An applicable real world scenario for this attack, for example, would be if an attacker set up a malicious access point with a similar SSID as a non-malicious access point in a public location, such as an airport or a coffee shop. The attacker could then host a server on his or her own personal computer that mimics a site where a user would enter sensitive information, like a banking site. The attacker could then manipulate DNS settings to instruct the malicious access point to forward all requests meant for the legitimate banking site

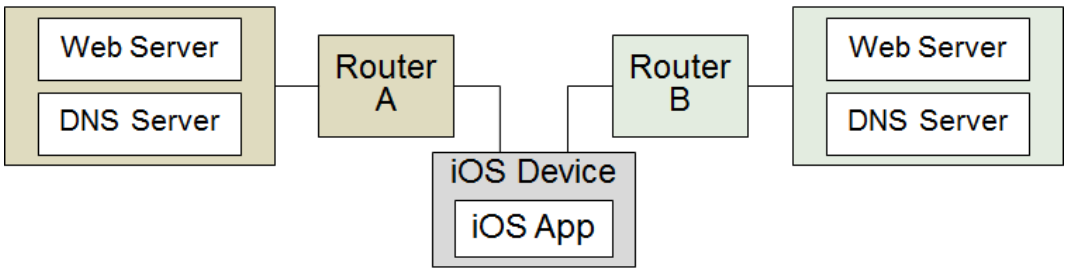

Figure 4.1: Attack Simulation Topology

to the fake banking site on his or her personal server where a user may then unsuspectingly enter sensitive information, such as log-in credentials. This would allow the attacker to then impersonate the user on the legitimate banking web site and cause harm.

4.1 Setup

There were several components needed to simulate the described attack; the main ones include two Linux virtual machines, each running Ubuntu version 12.04 and hosting both a web server and a DNS server; two wireless routers, a Rosewill RNX-N150RT wireless-N router and a Medialink MWN-WAPR150N wireless-N router; and an iOS device running our on-device application. In our setup the iOS device can connect to each of the wireless routers, which are connected to their own respective virtual machines hosting both a web server and a DNS server. A topographical diagram of our network setup can be seen in Figure 4.1.

In addition to setting up the components of our network in a specific way, we also needed to install specific software so that we could host a web server and a DNS server on each of the virtual machines. One program that we installed on both of our virtual machines is the Bind9 Domain Name System software. The Bind9 software allows us to set up our own domain name system and specify which domains resolve to

certain IP addresses by creating individual zone files referenced in the "named.conf" configuration file. We also installed the Apache2 web server software on both virtual machines in order to host web servers. After installing Apache2 we also created slightly different html pages for each web server which are served upon accessing the server. Once the DNS servers were set up on both virtual machines we needed to modify some of the settings on the routers. We set one of the routers to have the SSID "GoodInternet" and the other the SSID "Good1nternet" to portray a scenario in which a user connects to a malicious wireless access point attempting to mimic one that is not malicious. We also configured each wireless router to look to the DNS servers on their respective virtual machines as the primary DNS servers which we accomplished by specifying the address of the primary DNS server in the router settings. Once set up, the router will try to resolve the domain name of any request it receives by looking at the router's primary DNS server, which is the DNS server being hosted on the router's corresponding virtual machine. Once the router receives back from its DNS server the address that corresponds to the requested domain name, it passes the request on to the web server with that address, which incidentally is on the same virtual machine as the DNS server. The web server responds with its html page which then is sent back to the iOS device. This process is illustrated further in Figure 4.2.

4.2 Procedure

Simulating the attack after the setup is complete is a fairly involved process and requires some precision in its execution. The following is the procedure we followed in executing the attack simulation:

1. Run the "launcher.py" python program on the remote server.

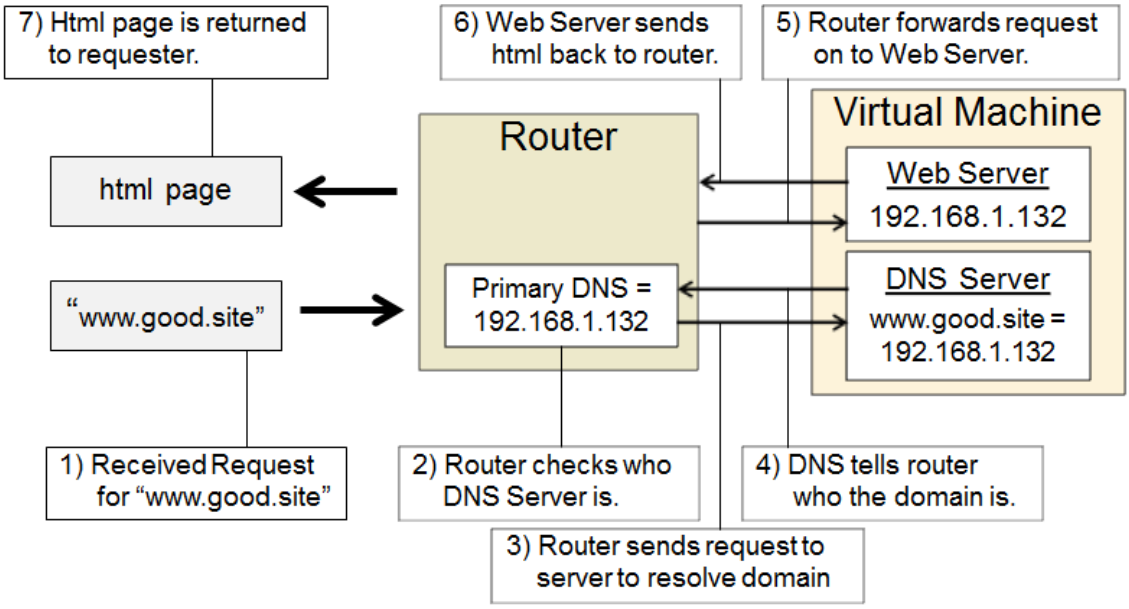

Figure 4.2: Request Path Traversal

2. Connect one of the virtual machines to the router with the SSID "GoodInternet" and the other virtual machine to the router with the SSID "Good1nternet."

3. On each virtual machine launch both the Apache2 Web Server and Bind9 DNS services.

4. Connect the iOS Device to the router with the SSID "GoodInternet."

5. Launch the on-device monitoring application on the iOS device.

6. Back out of the on-device monitoring application, leaving it to run in the background, and launch a web browser on the iOS device. In the browser navigate to the domain "www.good.site".

7. Refresh the page 2 to 3 more times, waiting a few seconds before each refresh.

8. Close the browser and in the Device settings on the iOS device, turn off the device's WiFi capability.

9. Repeat steps 6 - 8 two more times.

10. Now connect the iOS device to the router with the SSID "Good1nternet."

11. Launch a web browser on the device and navigate to "www.good.site".

12. Close the browser and in the device settings on the iOS device, turn off the device's WiFi capability.

13. Reconnect the iOS device to the router with the SSID "GoodInternet" and repeat steps 6 - 8 two more times.

14. Close the browser and in the device settings, connect to a WiFi access point with an internet connection to send all the logged data to the remote server.

Starting off, we run the "launcher.py" program to ensure that any data collected from the on-device application can be successfully received on our remote server. We then connect to the "GoodInternet" access point to try to simulate browsing on a normal, non-malicious network. In order to populate our data set with mostly non-malicious data, we disconnect from the non-malicious router and rejoin multiple times before connecting to the malicious router. Once we connect to the malicious router, we launch a web browser only once and navigate to what the user would think is the same web-page. We then immediately disconnect from that network and rejoin the non-malicious network. This will ensure that the malicious network does not appear as frequently in our resulting data as the non-malicious network, simulating how non-malicious networks are more common than malicious ones. At the end we connect to a network with an internet connection so the on-device application can send the collected information to our remote server.

To simulate a user unknowingly accessing a malicious web site, we created simple HTML pages that are returned for requests to both the malicious and non-malicious

Figure 4.3: Attack Simulation web pages

web servers. The pages are similar to each other in appearance but have slightly different text so that we could differentiate between them during the experiment. The benign and malicious web servers' HTML pages can be seen in Figure 4.3.

4.3 Results

Simulating the malicious router attack resulted in a data set that had 97 poll sequences spread over 10 different "Output" log files. We used our "logAnalyzer.py" program to sift through the data set and to demonstrate what kind of conclusions an analyst could potentially draw by using our analysis program on a data set of continuously collected information. Running our analysis program on the data without any input arguments resulted in the program outputting the data in a time-line table structure in which we could see during which sequence specific data elements were active. Assuming that an analyst looking at this data set would have no prior knowledge of the attack that was simulated, we attempt to replicate the steps that he or she could take to draw accurate conclusions about the malicious activity.

Without knowledge of which pieces of information in the resulting data set are pertinent, an analyst could start the analysis by first looking at the time-line data

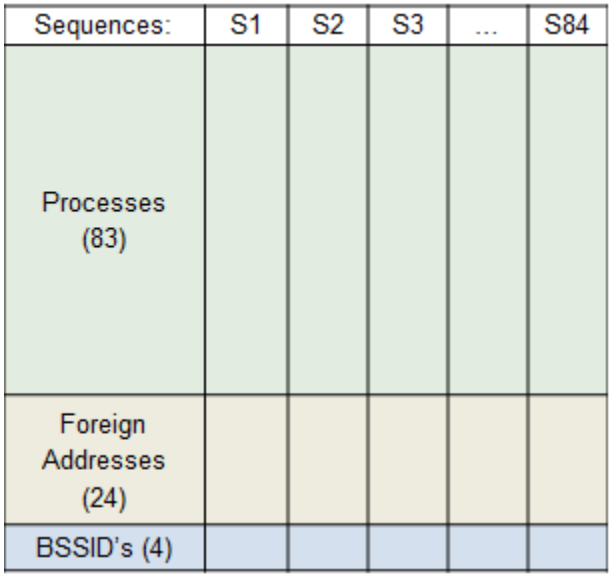

Figure 4.4: Attack Simulation Time-line Table Layout

structure, which would give him or her an unfiltered summary of all the data elements across every sequence in the monitoring session, showing which data elements were active at specific times. The time-line table of the data set for the simulated attack shows 111 data elements (rows) across 97 sequences (columns); however of those sequences, only the first 84 of them are actually relevant to the investigation since the last 13 occurred when the collected data was being sent to our remote server. Of the data elements, 83 of them describe the processes that were running on the mobile device during the monitoring session; 24 of them describe the foreign addresses of all the connections that were active during the monitoring session, and the remaining 4 data elements show the BSSIDs of the different network entries. This layout can be seen in Figure 4.4.

Because the time-line table structure is somewhat extensive it would be difficult for an analyst to determine that the device had connected to a malicious network only by looking at it. The next logical step would be for the analyst to try to figure out which

data elements are more interesting or suspicious than others, and this can be done by figuring out which data elements in the table correspond to any anomalous activities. To do this an analyst could modify the input arguments of the "logAnalyzer.py" program so that it also displays the Euclidean distance nearest neighbors result table for the k nearest neighbors, which in this case we will say is 5. Looking at the Euclidean distance nearest neighbors results table, the analyst would see several cells with asterisks appended to the distance values indicating that those sequences have anomalous distance values for that nearest neighbor. However, the cells containing anomalous distance values are fairly prevalent and spread throughout the table, so the table does not reveal any particular sequence to be terribly suspicious compared to the others. It is interesting to note, though, that with this data set, although the cells with the asterisks appended are prevalent throughout the table, several of those highlighted distances have a value of 0. The fact that the distances with a value of 0 are highlighted means that the average distance between sequences is larger than 0, but not necessarily that these sequences are suspicious. In fact, it means the opposite, since there would be at least k other sequences that are identical to the sequence in question. In contrast, a cell that is highlighted for having a Euclidean distance nearest neighbor value that is abnormally high would be suspicious since it would indicate that the sequence in question is not similar to any other sequence. Figure 4.5 is a screen shot of a segment of the Euclidean distance nearest neighbor output table, showing how some of the highlighted values are 0 and others are closer to 2.

An analyst may not be able to make any conclusions regarding anomalous activity by strictly looking at the Euclidean distance nearest neighbors result table since there are a lot of highlighted cells that are not actually indicative of anomalies. The analyst could, however, look at the Levenshtein distance result table to see if there are any correlations among the data elements by including the appropriate input argument

sequence 45	0.00 (seq 23)	0.00 (seq 46)	0.00 (seq 21)**	0.00 (seq 20)**	0.00 (seq 22)**
sequence 46	0.00 (seq 23)	0.00 (seq 21)	0.00 (seq 20)**	0.00 (seq 22)**	0.00 (seq 24)**
sequence 47	0.00 (seq 25)	0.00 (seq 27)	0.00 (seq 26)**	0.00 (seq 50)**	1.00 (seq 23)
sequence 48	0.00 (seq 49)	1.00 (seq 50)	1.00 (seq 62)	1.00 (seq 47)	1.00 (seq 61)
sequence 49	0.00 (seq 48)	1.00 (seq 50)	1.00 (seq 62)	1.00 (seq 47)	1.00 (seq 61)
sequence 50	0.00 (seq 25)	0.00 (seq 47)	0.00 (seq 27)**	0.00 (seq 26)**	1.00 (seq 23)
sequence 51	1.00 (seq 53)**	1.41 (seq 29)**	1.41 (seq 54)**	1.41 (seq 28)**	1.73 (seq 82)**
sequence 52	1.41 (seq 53)**	1.73 (seq 65)**	1.73 (seq 51)**	1.73 (seq 54)**	1.73 (seq 64)**
sequence 53	1.00 (seq 51)**	1.00 (seq 54)	1.41 (seq 52)**	1.41 (seq 55)**	1.41 (seq 63)
sequence 54	1.00 (seq 55)**	1.00 (seq 53)	1.41 (seq 51)**	1.73 (seq 37)**	1.73 (seq 63)**

Figure 4.5: Unfiltered Euclidean Distance Table Segment

when running the "logAnalyzer.py" program. The Levenshtein distance argument expects an integer value which the program uses as a maximum value for suspicious distances in the resulting table - highlighting with asterisks any values less than or equal to it. The resulting Levenshtein distance table for this data set shows a large number of highlighted cells, the vast majority of which are processes being compared with other processes. Furthermore, almost every process listed in the table has a Levenshtein distance value of 0 when compared to almost any other process. This means that almost all the processes are active and inactive at the exact same times across every sequence. An analyst could verify this by looking back at the time-line table structure and seeing that almost all the processes are in fact active across the entire data set. Because the active processes do not drastically change, the analyst could disregard them for the remainder of the analysis, specifying in the input arguments of the "logAnalyzer.py" program that only the network traffic information and network access point information should be considered.

Once the program is run again with the process information filtered out, the

sequence 45	0.00 (seq 16)	0.00 (seq 19)	0.00 (seq 49)	0.00 (seq 17)	0.00 (seq 23)
sequence 46	0.00 (seq 16)	0.00 (seq 19)	0.00 (seq 49)	0.00 (seq 17)	0.00 (seq 23)
sequence 47	0.00 (seq 16)	0.00 (seq 19)	0.00 (seq 49)	0.00 (seq 17)	0.00 (seq 23)
sequence 48	0.00 (seq 16)	0.00 (seq 19)	0.00 (seq 49)	0.00 (seq 17)	0.00 (seq 23)
sequence 49	0.00 (seq 16)	0.00 (seq 19)	0.00 (seq 17)	0.00 (seq 23)	0.00 (seq 62)
sequence 50	0.00 (seq 16)	0.00 (seq 19)	0.00 (seq 49)	0.00 (seq 17)	0.00 (seq 23)
sequence 51	0.00 (seq 55)	0.00 (seq 53)	0.00 (seq 54)	1.00 (seq 52)**	1.41 (seq 11)**
sequence 52	1.00 (seq 55)**	1.00 (seq 51)**	1.00 (seq 53)**	1.00 (seq 54)**	1.73 (seq 65)**
sequence 53	0.00 (seq 55)	0.00 (seq 54)	0.00 (seq 51)	1.00 (seq 52)**	1.41 (seq 11)**
sequence 54	0.00 (seq 55)	0.00 (seq 53)	0.00 (seq 51)	1.00 (seq 52)**	1.41 (seq 11)**

Figure 4.6: Filtered Euclidean Distance Table Segment

program displays a new time-line table, Euclidean distance nearest neighbors table, and Levenstein distance result table without any process information. This changes the Euclidean distance result table so that sequences that have a nearest neighbor value of 0 are no longer highlighted as being anomalous. The result is that now only a few sequences in the table have highlighted cells: sequence 15, sequence 41, and sequences 51 - 56; however, of these sequences, only sequences 52 had a highlighted cell for the nearest two neighbors, whereas all the other sequences that had highlighted cells only had them highlighted for their 3rd, 4th, and 5th nearest neighbors. What this means is that sequence 52 is more anomalous than any other sequence in the data set, since even the distances of the sequences closest to it were outside of standard deviation. These anomalies from the filtered data can be seen in Figure 4.6.

Knowing that sequence 52 is an anomalous sequence, an analyst would then be able to look back at the time-line table structure to investigate which data elements were active during sequence 52 and what that activity implies. As previously explained, the time-line table highlights any data elements (rows) that has an anomalous value

outside of standard deviation within any sequence that was determined to be anomalous in the Euclidean distance nearest neighbor result table. This means that an analyst could figure out which data elements are most suspicious within a sequence that was identified to be suspicious and then trace those elements through the time-line table to try to find any evidence of suspicious activity. In looking at the time-line table from our attack simulation with the processes filtered out, we see that four data elements have been highlighted: the foreign address "www.good.site.http", the network with BSSID "68:1c:a2:0:8c:d0," the newtork with no BSSID which instead says "NoConnection", and the network with BSSID "14:35:8b:8:4b:a0". Starting off, an analyst might look at the network with BSSID "68:1c:a2:0:8c:d0" since it is the data element that is first active among the four highlighted data elements. This network connection is active from sequence 0 through sequence 14. It then becomes inactive from sequence 15 through sequence 27 and then active again from sequence 28 through sequence 40, inactive again from sequence 41 through 62, and then active again from sequence 63 through sequence 83. Because our tool labels having no WiFi network connection as a type of network connection, whenever the network connection with BSSID "68:1c:a2:0:8c:d0" is inactive, another network connection is active. In particular, the network connection with BSSID "14:35:8b:8:4b:a0" is active from sequence 52 through sequence 55. The activity of the different WiFi networks can be seen in Figure 4.7.

These sequences also happen to be the sequences that were highlighted in the euclidean distance nearest neighbor result table, which could lead an analyst to conclude that this particular network connection is more suspicious than the others. Looking for any other activity that corresponds to this network, an analyst may notice that one of the other highlighted data elements, the foreign address "www.good.site.http," is only active in sequence 52 which is within our series of anomalous sequences. The

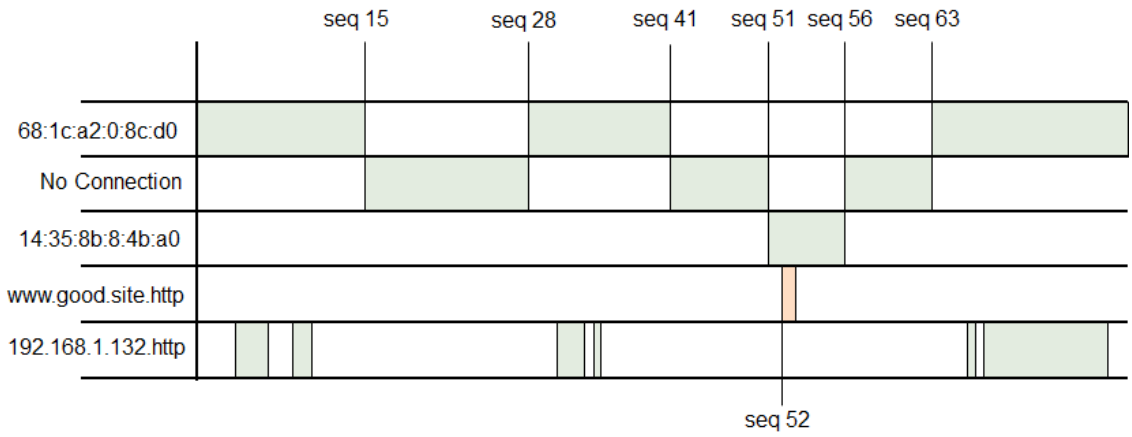

Figure 4.7: Attack Simulation WiFi Network Activity

fact that communication with a specific foreign address is the only network communication esatblished while on the network with the BSSID of "14:35:8b:8:4b:a0" is not necessarily incriminating for that WiFi network. What is incriminating, though, is the fact that the foreign address "www.good.site.http" was not active on any other networks or at any other place in the data set. This means that there is a strong possibility that that foreign address "www.good.site.http" is somehow connected to the network with BSSID "14:35:8b:8:4b:a0." This is evidenced further by the Levenshtein distance result table which shows that the two data elements have a Levensthein distance of only 4. The data set does show that other foreign addresses are exclusive to certain WiFi networks, such as the address "192.168.1.132.http" only appearing on the network with BSSID "68:1c:a2:0:8c:d0," but the difference is that that foreign address appears on that network multiple times, and the network itself is active throughout the data set, making it non-anomalous.

Seeing that the WiFi network with BSSID "14:35:8b:8:4b:a0" and the foreign address "www.good.site.http" are both anomalous and related, an analyst could then look at the actual log files themselves to see if any more information about these

```
Sequence Number: 39
Message Type: Network Nodes
2015-06-17 03:43:25
BSSID             SSID          IPAddress
68:1c:a2:0:8c:d0 GoodInternet PrivateIPAddress

 •••

Sequence Number: 52
Message Type: Network Nodes
2015-06-17 03:44:31
BSSID             SSID          IPAddress
14:35:8b:8:4b:a0 Good1nternet PrivateIPAddress
```

Figure 4.8: Excerpt from Log Files

data elements can be obtained. Knowing that the anomalous behavior occurs around sequence 52, an analyst can easily locate the pertinent information in the log files and find that the SSID of the WiFi network with the BSSID "14:35:8b:8:4b:a0" is "Good1nternet", and, in earlier log files, that the SSID of the WiFi network with the BSSID "68:1c:a2:0:8c:d0" is "GoodInternet". The analyst would notice that the SSID's of the two networks only differ by one character and that because the SSID of the anomalous WiFi network has the less conventional spelling of the same phrase, there is a good chance that it is attempting to mimic the non-anomalous network. Figure 4.8 shows an excerpt from the log files obtained through the attack simulation, highlighting the different SSIDs between a non-anomalous sequence and an anomalous one. Taking into consideration that there is a connection to a statistically anomalous foreign address on a statistically anomalous WiFi network that has an SSID suspiciously similar to a non-anomalous one, an analyst could conclude that there is a high probability that the device had connected to a malicious network.

Because we had a data set of continually collected information from the device, we showed how an analyst could use the "logAnalyzer.py" program to step through

the data set and eventually conclude that the device had connected to a malicious network. As we saw in our preliminary investigation, it is not possible to make that conclusion with the same level of confidence using a commercial forensics tool that can only get a static image of the device since, as previously mentioned, the information we had was somewhat limited. In the case of our preliminary investigation, we could see that one WiFi network adapter had an SSID that was similar to the other networks while having a different network adapter vendor and country code. The reason that this information is somewhat limited is because all of these details help describe the same idea, which is that the device had connected to multiple network adapters and one of them was suspiciously different from the others. In our simulated attack, an analyst could find similar information describing the same idea, that being that the SSID of one of the routers is suspiciously only slightly different than the SSID of the other router, but he or she could also find other information not directly related to that idea, which would expand the analyst's overall perspective, providing additional insight into determining what malicious events may have transpired. For instance, using the on-device application and the "logAnalyzer.py" program, we showed how an analyst could determine that in our attack simulation one of the routers was anomalous and also related to a connection to a web address that was anomalous. This is not to say, however, that the data collected from the commercial tool is not useful. It is important to note that although the on-device application was also able to collect information which led to investigating the idea that one of the network access points has some suspicious differences in addition to other ideas, it did not offer the same level of detail as the commercial forensics tool. The most accurate analysis, therefore, would take advantage of the detailed information that can be obtained from a commercial tool as well as the information about the device over time that comes from our on-device application.

Chapter 5

DISCUSSION

In our approach we show that continuously collecting information from a mobile device can significantly help a forensic analyst in an investigation; furthermore, we also show precisely how this data collection helps by simulating an attack on a mobile device and by using our analysis program to reason to an accurate conclusion. Our approach, though, is not without limitations, one of the most important of which is that our approach is only a proof of concept and not necessarily practically beneficial for a forensic analyst. The simulated attack that we created was designed to parallel a real-world scenario in the sense that the majority of the events in the data-set were non-malicious and a few anomalous events were malicious. An actual network-based attack carried out on an iOS device would likely be more complex than our simulated attack, and our approach may not be as effective at helping an analyst discover evidences of the attack. Additionally, we do not know for certain that our approach would helpful for developing countermeasures against the other attacks mentioned in this paper. If an analyst could collect enough information from a device to identify anomalous activities associated with a specific attack, such as with our own simulated attack, then he or she could begin devising incident response mechanisms as a countermeasure to that attack, but we cannot say for certain that our approach can identify those anomalous activities.

Another limitation of our approach is that the on-device application can only operate on the iOS platform. We originally chose to design our on-device application for the iOS platform since the original devices we were examining in our preliminary investigation were iOS devices. Unfortunately, many corporations use Android or

Blackberry devices for business-related communications, and with those corporations our on-device application would be inapplicable. Additionally, although we were able to successfully obtain various pieces of information from an iOS device to use for analysis, it is not guaranteed that the same information would be available on another platform, which also means that what we demonstrated through the simulated attack may not be as relevant. Additionally, because commercial forensics tools have different capabilities depending on the platform, we cannot guarantee that using an on-device application to collect information over time would provide the same benefit on other platforms as it did on iOS, since a commercial forensics tool could allow for an effective analysis of the device in and of itself.

The most notable limitation with our approach is with the amount of information we are able to collect using the on-device application. Naturally there are more areas of information that we would like to draw from using our on-device application, but we are unable to do so due to the restrictive nature of the iOS platform. The iOS platform implements a security feature known as application sandboxing, which according to Apple's documentation [21] is a kernel-level mechanism that restricts the data and resources that any third-party applications have access to. Specifically, applications that are sandboxed are unable to access data or resources that are particular to any other application, including first party applications. According to Bucicoiu et al. the sandboxing mechanism is primarily handled by a TrustedBSD mandatory access control module at the kernel level, which "enforces sandboxing at the level of system calls and directory paths" [22]. A good example of how this restriction limits our approach is with attempts to access SMS data. As previously mentioned, according to the paper by Han et al. some attacks on the iOS platform take advantage of the SMS service and attempt to charge the user premium rate messages without being detected [7]. If, however, we were able to keep track of the messages sent from the

SMS application and also see when the malicious SMS messages were sent by looking at the running processes, we could compare the time-stamps on the messages with the times that the SMS process was active and identify any malicious anomalies using our analysis program.

One possible solution to our problem of data restriction on iOS is to create additional applications and features that can be used to produce substitute or supplementary data for analysis. We can see how an additional application can help with analysis when considering our simulated SSID spoofing attack. When looking at the results, we can see that the device's connection to a specific foreign address was suspicious because the connection occurred on a network that was only active during a short period of time, which made it anomalous. Looking at the actual log files we can also see that the SSID of the suspicious network varies only slightly from the non-suspicious network, and from both of these pieces of evidence we can conclude that the anomalous network was set up in an attempt to spoof a legitimate network; however, we can reach this same conclusion with a higher level of confidence if we consider further evidence that corroborates it. One way to do this is to determine what URLs the user had entered when connecting to the foreign addresses. If we could show that the URL entered into the browser while connected to the first network was the same as the URL while connected to the second network, we could conclude with more confidence that the logs show evidence of an SSID spoof attack since the BSSID's of the network connections were different. Although we can see from the web browser screen shots that the user entered "http://www.good.site" into the search bar, only the foreign address of the second network connection shows evidence of that URL according to the log files as seen in Figure 5.1.

Developing another application to collect the entered URLs is a good way to provide the supplemental information that we would need to conclude with a higher

```
Sequence Number: 31
Message Type: Current Network Information
2015-06-17 03:42:45
Proto  Recv-Q  Send-Q  Local_Address        Foreign_Address      (state)
tcp4   0       0       192.168.1.139.50186  192.168.1.132.http   ESTABLISHED
tcp4   0       0       192.168.1.139.50185  192.168.1.132.http

Sequence Number: 52
Message Type: Current Network Information
2015-06-17 03:44:31
Proto  Recv-Q  Send-Q  Local_Address        Foreign_Address      (state)
tcp4   0       0       192.168.0.103.50189  www.good.site.http   ESTABLISHED
tcp4   0       0       192.168.0.103.50188  www.good.site.http   ESTABLISHED
```

Figure 5.1: Comparing Foreign Addresses of Different Network Connections

degree of confidence that the evidence shown in the analysis program is indicative of an SSID spoof attack. Due to the iOS sandbox policy, our on-device application that was used to continuously collect information in our attack simulation did not have access to the history of the browser that we used, and as a result, it did not log the URLs. Naturally, one would then question how an additional application would be able to obtain the data that the data collection application could not. Simply put, it cannot since it falls under the same restrictions as the first application; however, because we are designing a proof-of-concept system intended to aid companies in securing the data on their mobile devices, we have some freedom in dictating how those devices should be secured, meaning we can suggest policies for device usage that would accompany our design for security. If then we create our own web browser application, we could both gain additional information from the browser and ensure that the users would actually use it, since we could recommend its usage as part of a security policy.

Although it would be difficult to develop a web browser with the same functionality as those commonly used on iOS such as Safari or Google Chrome, it is possible to develop a rudimentary browser with basic security features. We developed such

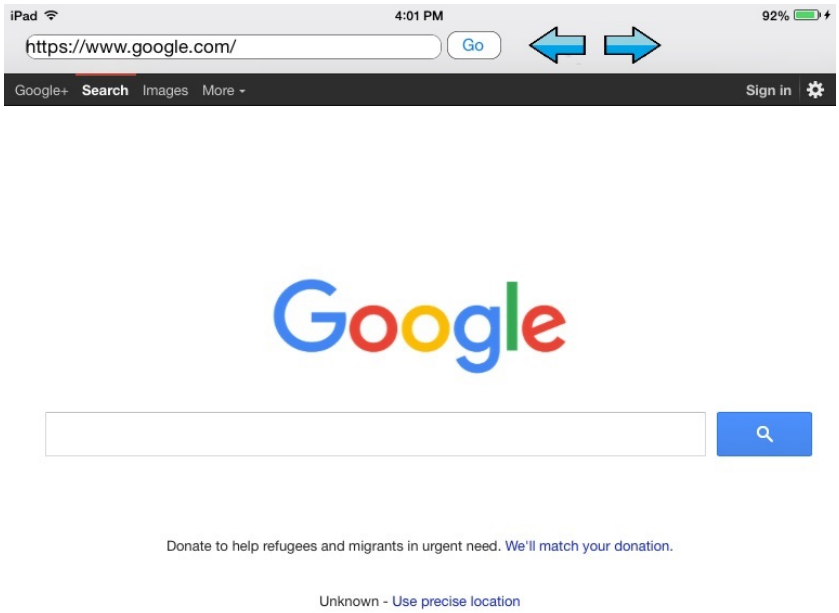

Figure 5.2: Browser Application Screen Shot

a browser application, and although we never used it when carrying out the simulated attack, we can still describe its effectiveness when used alongside our on-device collection application. Figure 5.2 shows a screen shot of our browser application.

If we know ahead of time that the user will use our browser application instead of a commercial browser, then we can program functionality that would send any information private to the application to our remote server, including the browser history. Had this been the case with our simulated attack, we would have been able to determine that the URL of the foreign address on the first network matched the URL of the foreign address on the second network, even though the foreign addresses were different. Considering this along with the facts that the two networks had different BSSIDs, similar SSIDs, and that one network was related to anomalous browsing activity, an analyst could have determined with an even higher level of confidence that the simulated attack was an attempt to spoof the SSID of a non-malicious network.

In regards to future work, our approach can be expanded upon by improving our on-device application to continuously collect more types of information, such as SMS information or device settings information. Porting the application to other mobile platforms could also prove useful as it would allow for the exploration of the effectiveness of continual data collection on platforms besides iOS. Through our analysis program, we demonstrated how Euclidean distance and Levenshtein distance calculations can be used to aid a forensic analysis, but there may be other calculations that can be made that take advantage of the additional context gained from continuous data collection. Adding more calculations to the analysis program would only improve the level of confidence with which an analyst could make conclusions. Furthermore, we used a single simulated attack to demonstrate the effectiveness of our approach. It stands to reason, therefore, that simulations of other types of attacks or real-world usage could help improve our tools by demonstrating their effectiveness even further or highlighting areas in which they can be improved.

We can see that there is potential for expanding our approach through the addition or improvement of specific features in our iOS application; however we can also expand our approach by considering implementation on other platforms, such as Android. Naturally one of the biggest concerns we would have when considering porting our application to another platform would be whether or not we could achieve the same goals that we can achieve with our iOS implementation; however, because Android has fewer restrictive security features than iOS, we would likely be able to continuously collect a sufficient amount of information to effectively assist a forensic analyst. This can be seen in the paper by paper by Justin Grover in which he details how his Android monitoring application is able to collect a variety of different types of data including, but not limited to, application installation times, browser history, browser searches, calendar events, call logs, contacts, location settings, incoming and

outgoing SMS messages, and information from third-party logging applications [12]. According to the paper, an Android monitoring application can in fact gather an even greater variety of different types of information than an iOS monitoring application. Like iOS, Android also has a sandbox security feature that inherently restricts an application's access to assets from another application unless explicitly given. The difference, though, is that there are many more permissions available to an Android application than there are to an iOS application. One example is that applications on Android can request permissions to a specific web browser application upon installation which would allow the application to access browser history and browser search information as demonstrated in [12]. Additionally Android allows developers to specify permissions regarding their own applications which other applications can request to gain access to assets from those applications [23]. This feature would then allow applications to collaborate their data locally instead of needing to use an outside server, which was the case with our application on iOS. Overall it seems that in terms of data collection, the Android platform is suited for performing all the tasks that our on-device application performs on iOS and more.

Chapter 6

CONCLUSION

We have developed an on-device application for the iOS platform to continuously collect information about the state of the device, such as current network connections and active processes, every few seconds. The information collected is sent to a remote server where it is separated into log files to be used with the analysis program we created. The analysis program can perform correlation and anomaly distance calculations on the collected data that can help an analyst in drawing conclusions about whether or not any malicious activity had occurred on the device. We also simulated a potential network attack on a test iOS device running our application and demonstrated how an analyst could use the tools we developed, in addition to commercial forensics tools, to efficiently reason to an accurate conclusion.

The primary contribution of this research thesis is that it shows how continuously collecting information from a mobile device can lead to a more complete set of evidence which can then be used to improve a forensic analysis. We initially demonstrate the usefulness of a forensic analysis in regards to securing data on mobile devices by highlighting some of the weaknesses in modern Mobile Device Management systems, particularly their inability to detect intrusions or compromises. To support this claim we reference various papers that describe attacks carried out against mobile systems and give hypothetical procedures as to how an analyst could use forensics to aid in the discovery of those attacks. We continue to explore the effectiveness of forensic techniques in regards to mobile security by conducting a preliminary forensic investigation from which we find that traditional commercial forensic tools, although useful, do not necessarily give an analyst enough information to draw a conclusion with a

high level of confidence. Supplementing this information with continuously collected information from the device, however, makes the evidence more complete by adding more context for it, since an analyst can now know how different parts of the evidence relate to each other over time. Although other works have suggested that having more context of the collected information can lead to more accurate conclusions, we show precisely how this can be accomplished by using distance calculations to correlate different data elements and detect anomalous data, and we demonstrate its effectiveness with a simulated data set.

Regarding the future work of our approach, we have shown how, like with similar applications on other mobile platforms, there is still the potential to collect more types of data to be used in a forensic analysis. Naturally, the restrictive nature of the iOS platform could make this advancement challenging, but not impossible if we can use additional applications to collect data. Additionally, it is also feasible that we would be able to port our approach over to the Android platform. Some monitoring applications similar to ours already exist on Android, but unlike existing applications, our approach uses a sophisticated analysis tool to produce statistics useful for drawing conclusions with a high level of confidence.

REFERENCES

[1] Daniel Brodie. Practical attacks against mobile device management (mdm). https://media.blackhat.com/eu-13/briefings/Brodie/bh-eu-13-lacoon-attacks-mdm-brodie-wp.pdf).

[2] Rashid Fahmida. Enterprises Need More Than MDM to Address Mobile Security Risks: Analysis.

[3] Andrew R Scholnick. Facilitating forensics in the mobile millennium through proactive enterprise security. In *Proceedings of the Conference on Digital Forensics, Security and Law*, pages 141–154, 2012.

[4] Edward Kovacs. Apple Working to Patch Gatekeeper Bypass Flaw. http://www.securityweek.com/apple-working-patch-gatekeeper-bypass-flaw.

[5] Justin Paglierani, Mike Mabey, and Gail-Joon Ahn. Towards comprehensive and collaborative forensics on email evidence. In *Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom), 2013 9th International Conference Conference on*, pages 11–20. IEEE, 2013.

[6] Billy Lau, Yeongjin Jang, Chengyu Song, Tielei Wang, PH Chung, and P Royal. Mactans: Injecting malware into ios devices via malicious chargers. *Proceedings of Black Hat USA*, 2013.

[7] Jin Han, Su Mon Kywe, Qiang Yan, Feng Bao, Robert Deng, Debin Gao, Yingjiu Li, and Jianying Zhou. Launching generic attacks on ios with approved third-party applications. In *Applied Cryptography and Network Security*, pages 272–289. Springer, 2013.

[8] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. Jekyll on ios: When benign apps become evil. In *Usenix Security*, volume 13, 2013.

[9] Aldo Cassola, William K Robertson, Engin Kirda, and Guevara Noubir. A practical, targeted, and stealthy attack against wpa enterprise authentication. In *NDSS*, 2013.

[10] Antonio Bianchi, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Blacksheep: detecting compromised hosts in homogeneous crowds. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 341–352. ACM, 2012.

[11] Lee Reiber. Mobile Security for a Nomadic Workforce. http://www.infosecinsight.com/2014/06/mobile-security-nomadic-workforce#.VcVOzXFVhBc.

[12] Justin Grover. Android forensics: Automated data collection and reporting from a mobile device. *Digital Investigation*, 10:S12–S20, 2013.

[13] App Programming Guide for iOS. https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html.

[14] API. http://flask.pocoo.org/docs/0.10/api/.

[15] Bjarne Mangnes. The use of levenshtein distance in computer forensics. 2005.

[16] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. Efficient algorithms for mining outliers from large data sets. In *ACM SIGMOD Record*, volume 29, pages 427–438. ACM, 2000.

[17] Aleksandar Lazarevic, Levent Ertöz, Vipin Kumar, Aysel Ozgur, and Jaideep Srivastava. A comparative study of anomaly detection schemes in network intrusion detection. In *SDM*, pages 25–36. SIAM, 2003.

[18] Riyanat Shittu, Alex Healing, Robert Ghanea-Hercock, Robin Bloomfield, and Muttukrishnan Rajarajan. Intrusion alert prioritisation and attack detection using post-correlation analysis. *Computers & Security*, 50:1–15, 2015.

[19] Ball Tree Explained in a Simple Manner. https://ashokharnal.wordpress.com/tag/ball-tree-explained-in-simple-manner/.

[20] Wikipedia. Ball tree — wikipedia, the free encyclopedia, 2015. [Online; accessed 8-August-2015].

[21] App Sandbox Design Guide. https://developer.apple.com/library/mac/documentation/Security/Conceptual/AppSandboxDesignGuide/AboutAppSandbox/AboutAppSandbox.html.

[22] Mihai Bucicoiu, Lucas Davi, Razvan Deaconescu, and Ahmad-Reza Sadeghi. Xios: Extended application sandboxing on ios. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 43–54. ACM, 2015.

[23] System Permissions. http://developer.android.com/guide/topics/security/permissions.html#arch.