MALICIOUS APPS MAY EXPLOIT SMARTPHONE'S VULNERABILITIES TO DETECT USER ACTIVITIES

A Thesis Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Xi Lu

May 2017

MALICIOUS APPS MAY EXPLOIT SMARTPHONE'S VULNERABILITIES TO DETECT USER ACTIVITIES

Xi Lu

APPROVED:

MALICIOUS APPS MAY EXPLOIT SMARTPHONE'S VULNERABILITIES TO DETECT USER ACTIVITIES

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Xi Lu

May 2017

Abstract

In order to make apps functional, mobile operating systems, such as Android, allow applications to access some system data without asking for user permission. We demonstrate that by analyzing these system data and some side channel information, it is possible to gain insight into a smartphone user's behavior, thus putting their privacy at risk. With these real-time privacy information collected, a malicious attacker may launch spear phishing attacks with much higher yield rates. In this thesis, we study a combination of power consumption, network traffic, and memory usage of several commonly used activities, and demonstrated that it is possible to classify a user's smartphone activities into one of six categories, which are Video, Game, Internet, Music, Idle, and Phone Call. We designed several experiments to test the classification which resulted in high success rates. We also present the possibility of detecting transitions of smartphone activities.

Contents

1 Introduction **1**

2 Prior Work **12**

2.1 Side Channels and Information Leaking 12

2.2 DDoS and TDoS 14

3 Background **16**

3.1 Scenario Definition 16

3.2 Side Channels of Android OS 17

3.3 Decision Tree 22

4 Classification Methodology **24**

4.1 Single Scenario Classification 24

4.2 Scenario Transition Detection 36

5 Experiment Design and Results **41**

5.1 Building the Decision Tree 41

5.2 Testing on Real Devices 44

5.2.1 Single Scenario Classification 45

5.2.2 Change Labeling of Decision Tree 46

5.2.3 Scenario Transition Detection 47

5.3 Data Source Robustness Analysis 49

6 Conclusion and Future Work 51

List of Figures

4.1 Power consumption examples of the six different scenarios. 27

4.2 Sample Internet traffic load in various types of scenarios. 28

4.3 The changing trend of power consumption of six different scenarios. . 29

4.4 The changing trend of incoming network traffic of six different scenarios. 29

4.5 Decision tree before applying training data 34

4.6 An illustration of using shifting base windows and increments for detection. 38

4.7 Relationship between accuracy rate and base-window size 39

4.8 Relationship between accuracy rate and maximum-window size 39

5.1 Training cases distribution for end nodes in first round of cross validation(P: Power, T: Traffic, A: Audio, X: Idle, M: Music, C: Phone Call, G: Game, I: Internet, V: Video) 42

5.2 Training cases distribution for end nodes in second round cross validation(P: Power, T: Traffic, A: Audio, X: Idle, M: Music, C: Phone Call, G: Game, I: Internet, V: Video) 43

5.3 Decision Tree Used for Scenario Detection (P: Power, T: Traffic, A: Audio, X: Idle, M: Music, C: Phone Call, G: Game, I: Internet, V: Video, *: unable to classify) 44

5.4 Altered Decision Tree Used for Scenario Detection (P: Power, T: Traffic, A: Audio, X: Idle, M: Music, C: Phone Call, G: Game, I: Internet, V: Video, *: unable to classify) 46

List of Tables

4.1 A sample data clip of side channel data in Video scenario 25

4.2 Attributes used for analysis and their Status Codes (SC) 33

5.1 Percentage of different scenarios' training cases falling into end nodes 42

5.2 Distribution of test cases of six scenarios and average classification accuracy rates 46

5.3 Updated distribution table of test cases 47

5.4 Border detection success rates and classification accuracy of 5 scenario transitions (with 5 test cases each) 48

Chapter 1

Introduction

In this digital age, people are heavily dependent on mobile devices for voice, video, and data communication, and they store a significant amount of sensitive information on their device. Preventing private information leakage from smartphones has become an important issue. Sandboxing is used to prevent unauthorized access to information gathered by other applications [19] ; however, smartphones know much more about their owners than they may realize. Leaking user behavior and private data represents a serious security threat to users, as smartphones are almost always connected to cell phone and/or Wi-Fi networks.

This thesis explores whether it is possible for a malicious app on a user's device to monitor activities the user is engaged in. There are certain smartphone software apps that can be installed to monitor, for example, a child's phone usage; since such a scenario would require the consent of the owner to be installed, this type of app is not the focus of our study. Here, we assume that the user is unaware of "malware" installed by hackers, and that the malware has the same access privileges to system information as any other app.

Smartphones are also vulnerable to Distrubuted Denial of Service (DDoS) attacks. In a DDoS attack, multiple compromised devices make requests towards a single system or a server, absorbing the resources of the targeted system. Compromised devices are often infected by viruses, such as Trojans. The main difference between DDoS and Denial of Service (DoS) attacks is that the attacker launching a DoS attack usually uses only one computer, while multiple devices are involved in DDoS. Several different kinds of DDoS attacks have been described by researchers. Karami, Park, and McCoy, 2016 discussed the DDoS-for-hire business [16], in which profit-motivated adversaries are available to scale up their networks to perform DDoS attacks; this type of attack has caused significant damage to public services such as the Microsoft Xbox and the Sony PlayStation networks. DDoS attacks can also utilize internet protocols. Czyz et al., 2014 described the rise and decline of DDoS attacks using NTP [9].

With IP spoofing technology, attackers are able to use devices running certain protocols, such as DNS and NTP, as amplifiers. When launching an attack, the attacker uses a spoofed IP address to send a query, and the responses from devices running the protocol will be sent to the spoofed address, which most times is the victim. Such attacks are identified as amplified DDoS attacks. A similar method can also be applied on DNS, which is a much more widely installed protocol. Rijswijk-Deij, Sperotto, and Pras, 2014 performed a comprehensive measurement study for DNSSEC and its potential for DDoS attacks [32]. When an attacker uses spoofed

IP information to send queries to open DNS resolvers, responses will be sent to the spoofed victim's IP address and generate a large amount of traffic. This kind of attack is also called amplified DDoS attacks. Devices sending responses to victims are called amplifiers. Notice that those devices are not compromised in an amplified DDoS attack; rather, it is an abuse of network protocol. This factor makes such an attack extremely hard to detect and defend against. Smartphones, which are running these protocols as well, are also targets of such attacks. Moreover, smartphones have fewer resources compared to commercial servers, so they are particularly vulnerable to this kind of attack.

Cisco, 2014 published a technical report about Telephony Denial of Service, known as TDoS [8]; this is an attack type that specifically targets phones. The TDoS attack pattern is similar to that of the traditional DoS. The attacker uses one or multiple compromised devices to launch call flooding to a target device, causing disturbances or exhausting resources. Such calls are called robocalls. Currently, TDoS is becoming an entire industry. With an automated calls generator, an attacker can easily generate thousands of robocalls. Those calls can then be directed to a single number. Such a large amount of traffic would cause an immediate DoS. Moreover, one robocall can last for a long time, which makes TDoS attacks more dangerous. If the attack succeeds in compromising mobile devices with backdoor malware, it is a distributed TDoS. Under this situation, robocalls can come from a calling center located anywhere; this increases the difficulty of tracing such an attack. By launching a TDoS attack, an attacker can prevent legitimate users and customers from using a targeted system's voice network. For institutes whose profits depend on functioning voice networks, TDoS attacks can cause significant financial loss.

As Voice over IP (VoIP) networks have increased in use, TDoS attacks have become easier to launch. The vulnerable part of a VoIP network is the Public Switched Telephone Network (PSTN). This used to be a closed network; however, with the use of VoIP, it has become more and more similar to the Internet. This is a security risk because it is easier for an attacker to launch large malicious or junk calls into a PSTN. Endpoint users have no control against PSTN migrating to an IP. This trend, combined with the increase in smartphones' market share, makes TDoS attacks a significant threat.

Since telephony networks are adopting Internet protocols, attackers now can launch TDoS attacks using traditional DDoS methods. According to Cisco's report, attackers can automatically generate a large amount of calls in a few days using tools [8]. One way to detect such attacks is to trace back each call to its original carrier; However, this method is not applicable for distributed TDoS attacks. As already mentioned, compromised devices can be located anywhere. By compromising smartphones, TDoS attacks are even harder to detect, since each calling device is an actual user's phone. Even if device owners can be tracked down, it does not mean they are the actual attackers.

Social networks, such as Twitter and Facebook, also increase the risk of large-scale TDoS attacks. Malicious scripts used for compromising mobile devices can be easily spread through social networks. When users on smartphones browse social networks, their phones are vulnerable to these malicious scripts. Another way to launch an attack is to use the power of a large number of followers. Social network users with large numbers of followers can deliberately or inadvertently spread malicious programs to multiple devices, even directly requesting followers to call a single number. This is an example of a TDoS attack. In this thesis, we will focus on the use of malicious scripts.

Recently, a potential TDoS attack on the 911 system appeared in the media [35]. The attacker posted a link on Twitter and used an account with a large number of followers to encourage people to click and spread the link. The link, in fact, was a malicious Javascript script that forced infected smartphones to dial 911 repeatedly. By utilizing an iOS vulnerability, iOS device users who clicked the link were taken to the malicious script. When their browser downloaded the script and ran it on their device, the device repeatedly dialed 911. Fortunately, there were not enough smartphones compromised to cause significant disruption. However, such malicious scripts are undeniably a potential threat. If the number of compromised smartphones were large enough, it would become a TDoS attack on the 911 system. According to the Cisco TDoS report [8], 500,000 calls in a short time would be able to shut down emergency services.

The TDoS attack just described aimed at emergency system targets of iOS devices; however, with Android, such attacks are even easier. Most users do not know much about Android's permission system; thus, it is very likely for them to grant too much permission to an application. With enough permission to monitoring the system, a malicious application running in the background can stealthily make phone calls without the user noticing. By rendering the device into a SIM-free state and with IMEI spoofing, researchers have found that such attacks can be undetectable and unable to be defended against in the short term. Because it is not feasible to disable SIM-free emergency call, such attacks are even more harder to defend against. Even if without SIM-free mechanics, unsolicited calls are still possible. Such stealthy and illegal calls can be directed not only to emergency systems, but also to premium numbers, which would cause monetary loss.

TDoS attacks can also cause damage to big enterprises, especially enterprises that rely on telephone networks to communicate and operate. In a short time, a TDoS attack can cause a large financial loss. TDoS can also be used for toll fraud and personal harassment. These kinds of attacks should give us pause about smart device security. Compromised devices may perform tasks in the background without users noticing it. Our study provides insights into detecting such stealthy and illicit behavior of possibly compromised devices.

The Android operating system, which is open-source software and can easily be modified. It has gained a large market share among smart phone platforms [14]. However, precisely due to Android's openness, it has become a principle target of malware attacks. Even though queries require the user's permission during installation, sensitive data, including the user's precise location, can be obtained by International Mobile Station Equipment Identity (IMEI), and carrier information can be obtained through software. However, power consumption, audio status, Internet traffic, and memory information under `/proc/[pid]/statm` are usually not considered to be sensitive; they can be accessed through API and parsing kernel files.

We will demonstrate that by reading power usage, audio interface, network statistics, and memory usage of system processes, a user's behavior can be inferred on the

Android platform. Such kind of data sources are called side channels. All such information can be accessed without any special user permission. By analyzing data using machine learning techniques, we have been able to determine whether a user is watching a video, browsing web pages, listening to music, or putting their phone on idle. This approach does not require prior knowledge, for example, of user habits or a currently running application list.

Android has included a permission or notification system to limit access to sensitive information and related API access by apps. However, a device can leak a user's activities from the information shared by the phone's OS or from other side channels. In order for an app on a smart device to work, it must access system information. Seemingly harmless information may be accessed without explicit user permission or notification. For example, PowerSpy was able track one's location by analyzing the power consumption of one's mobile device [21].

Android side channels have been exploited by many researchers before. Battery information about power consumption is generally grouped as power side channels. Information from the cache indicating memory usage by processes or thread are grouped as cache side channels. Other new side channels include accelerometers, barometers, and gyroscopes. With the development of new hardware technology, more and more sensors are being added into smartphones. Data within most of these devices are available to internal programs without any special permission from users. This provides a possible way of leaking information. The software app mentioned before, An application, called PowerSpy [21], is able to detect a user's daily commute routine with rough GPS data and battery life information. While keystrokes cannot be directly recorded from the screen, they can be inferred from accelerometer data. Currently, such readings are of coarse value since mobile sensors are limited by their precise level. In the future, however, we will see more diverse and precise sensors equipped on smartphones, which will increase the likelihood of privacy leaks.

There are several ways to access power side channels on Android. Android has its own API to determine power usage, and such kind of information can also be gained from hardware measurements. By directly connecting chips or other parts of devices with outside hardware, one can get precise values for each part's power consumption; however, this approach is seldom used in side channel attacks since it requires a physical connection between victim devices and measurement hardware. Therefore, in this work, we only discuss power side channels gained from the Android system. By leveraging cache side channels, a hacker can obtain valuable information about foreground or background processes. It is possible to utilize cache side channel to infer the user interface of a foreground app or to infer what webpage a user has browsed. We will discuss these methods further in Chapter 2.

By leaking side channel information, an attacker can infer a user's smartphone usage routine. If a certain scenario is present, such as surfing the Internet, the attacker can utilize sensors, such as accelerometers or soft keyboard deployment monitoring, to get private information, such as keystrokes. Such a malicious application may provide hackers a means to engage in spear phishing. Spear phishing is a kind of accurate phishing targeting a certain user. The difference between spear phishing and normal phishing emails is that spear phishing usually requires some information of the target. Normal phishing emails are automatically generated and sent randomly. However, spear phishing targets certain kinds of users by exploiting known information from the potential victim. For example, if an automobile dealer or real estate agency accidentally leak a customer's contact information, the customer may receive phishing emails specifically about buying a car or home. Comparing to normal phishing, spear phishing has a higher success rate, but requires more information about the target. By utilizing this kind of side channel leaking, an attacker can gain detailed information about an intended victim in a timely manner, which greatly enhances success rates of phishing attacks. If an attacker knows the user is watching a video or listening to music at certain time, the attacker can send phishing emails pretending to be movie or music introductions. It is also possible for the attacker to infer a user's habits of interacting with their smart devices. The combination of such kind of information with social network analysis could even enable the attacker to discover the identity of the user.

We developed a method to discover a user's mobile phone activity without asking special permission from the user. Malicious applications like this may be used for spear fishing and lead to financial and information loss. All data we obtained were considered non-sensitive by security tools on the market. We managed to detect user behavior with a relatively small delay time of two minutes, while continuously monitoring change. Our approach shows it is possible to retrieve near real-time user-activity information. While all of our computations were done on our remote server, with an enhanced mobile processor, it would be feasible to manage all data processing on actual smartphones. Since communication with a remote server is less common, it's harder for firewalls that monitor outcome traffic to detect malware running in background.

Our approach provides a way of monitoring an intended victim's smartphone behavior without requiring much information about the user. Our goal was to detect a user's behavior with a maximum latency of two minutes. We accessed power usage data and incoming network traffic data once every ten seconds. We sent the collected data to our remote server for analysis once every two minutes to avoid high-resource consumption and to increase stealth. With a more powerful mobile system, it is conceivable that the analysis would not have to be done offline. We also monitored whether audio media were active, in order to know if a media player was running. These features were used to distinguish various user activities. Our system also retrieved the shared virtual memory usage data of `SurfaceFlinger`, the system process responsible for graphics on every Android device. By recording trends of this feature and monitoring fluctuations, we were able to determine whether a user was switching between different applications. We also monitored screen status changes.

In this thesis, we show that side-channel information may enable privacy leaks. The side-channel information discussed in this thesis does not require special permission from users; thus, the malware is able to collect data in a stealthy manner. With classification techniques, an attacker is able to infer users' interactions with their smartphone.

The layout of this thesis is summarized as follows. In Chapter 2, we review the state of the art in the field of smartphone malware and summarize the technical information of the Android system needed for our detection system. In Chapter 3, we describe the side channels we used in this work and some modules of the Android system we used. We described our algorithm of scenario classification in Chapter 4. We present our test results and analysis in Chapter 5. Finally, we conclude in Chapter 6.

Chapter 2

Prior Work

2.1 Side Channels and Information Leaking

Information leaked from side channels may lead to side channel attacks. Yan et al., 2015 [34] provides a detailed analysis of Android system side channels. Chen et al., 2014 [7] used shared virtual memory as a side channel to peek into applications running in the foreground without actually breaking into the Android sandbox mechanism. Saravanan et al., 2014 [28] and Zhang et al., 2009 [36] kept logs of users' key strokes and managed to identify users by their interactive touch screen habits. Andrioths et al., 2013 [1] further analyzed this approach and compared pattern screen-lock and traditional PIN screen-lock. Orthacker et al., 2012 [23] analyzed the Android permission system and stated that misunderstanding the meaning of permissions and a combination of permissions can make Android devices vulnerable to attackers. Aviv et al., 2012 [2] investigated the practicality of using an accelerometer as a side channel for the Android system.

Michalevsky et al., 2015 [21] looked into the relation between battery draining speed and a device's distance from its base station, and were able to infer a device's location. Jana and Shmatikov, 2012 [15] tracked memory usage inside browser processes, and used this as a fingerprint to guess a user's browsing behavior. Bianchi et al., 2015 [5] combined several side channels, such as inside process memory usage and audio interface, and demonstrated a novel GUI attack. Zhou et al., 2013 [38] looked into information leakage from public resources and how to relates it to the Android system. They presented a case in which it was possible to detect a user's identity using combined information from public resources and Android side channels. Michalevsky, Boneh and Nakibly, 2014 [20] discovered that with gyroscopes equipped on smartphones and some hardware, it is possible to recognizing speech.

Our approach utilized multiple side channels and constructed a decision tree from this information. While previous researchers presented approaches requiring prior knowledge of an intended victim's habits, such as an application installation list of the device, daily commute routine. our work shows that it is possible to derive an intended victim's activity without first knowing much about that user's habits.

Power consumption information has been commonly used as a side channel. Zhang et al., 2010 [37] presented a detailed and fine-grained battery usage analysis for several popular Android devices. Pathak et al., 2012 [24] went further to develop software called Eprof which measures battery usage. The power-usage models in these papers formed the basis of our approach on this topic.

Hornyack et al., 2011 [13] showed interesting results by analyzing common behaviors of malware, and developed a system that runs in the background that involved "pausing a suspicious process" when sensitive information was presented in the foreground. Grace et al., 2012 [11] looked into the advertisement library and its behavior in popular applications. One of the goals of our approach was to keep our monitoring stealthy by lowering the sampling rate and requesting as few permissions as possible.

2.2 DDoS and TDoS

Kuhrer et al., 2014 [17] described the method of an amplified DDoS attack and proposed a way to reduce its effects. In their paper, such attack's threat model is explained in detail. An Internet-level scan was performed to get reliable data. In another paper by Kuhrer et al., 2014 [18] the authors state that it is possible to abuse TCP to launch an amplified DDoS attack. Paxon, 2001 [25] provided a detailed analysis of the amplified DDoS attack's nature and suggested possible ways to defend against it. Buscher and Holz, 2012 [6] looked into botnets (compromised devices) used in DDoS attack. Bailey et al., 2005 [3] provided a monitoring system to investigate the IP infrastructure of the Internet to prevent IP address abuse.

Guri, Mirsky and Elovici, 2016 [12] looked into the potential threats that a compromised smartphone can pose to the emergency 911 system. It is possible for a smartphone to make a phone call without its user noticing, and such a phone call is extremely stealthy. The author states that it is possible to render smartphones into a SIM-free condition without taking the SIM card out. In this condition, calling to emergency center is permitted but unable to be traced. Pelechrinis et al., 2011 [26] described a possible mobile network DDoS attack in which the attacker is able to jam wireless networks by abusing protocol parameters. Becher et al., 2011 [4] gave a definition of mobile security and analyzed different approach of attacks. Seo, Lee and Yim, 2012 [30] proposed a detection framework for Android malware. Seo et al., 2014 [29] investigated the feasibility of smartphones compromised as botnets and the possibility of launching an attack that threatrns homeland security. Suarez-Tangil et al., 2014 [31] provided a way to detect Android malware by analyzing code structure.

Chapter 3

Background

3.1 Scenario Definition

In our study, we divided a typical smartphone user's activities into six categories, called scenarios, as follows:

- Idle is the idle state of the user's device, which means the device is not doing anything in the background or foreground. In our experiment, we disabled all applicable processes; however, some system processes could not and should not be terminated.
- Video contains all user activities related to displaying video online, regardless of the application, including video through a browser or YouTube.
- Game contains all user activities related to playing games; for our purposes, a game is not played online but it may upload and download a small amount of

data for advertisements.

- Internet contains all user activities related to surfing the Internet, excluding videos and music; Internet-related activity through browsers or through specialized applications such as Facebook, Twitter, and Instagram are all considered Internet scenarios.

- Music contains all user behavior related to listening to music; downloading process via Wi-Fi is the most common real-world method, and only takes several seconds; thus, it was unable to be detected in our current sampling. Therefore, in our measurements, we treated local and online music listening identically.

- Phone Call contains all user activity related to making audio phone calls, regardless of whether a call is routed through Wi-Fi or a carrier's network.

These six scenarios were our basic (atomic) scenarios. We defined a composite scenario as two or more basic scenarios overlapping at the same time; in this case, one scenario of the two will be in background. In this project, we considered two composite scenarios: "Internet + Music" and "Internet + Phone Call". The former was defined as listening to local music while browsing the Internet. The latter was defined as making a phone call while browsing the Internet.

3.2 Side Channels of Android OS

Side channels are data sources used by an attacker to infer more sensitive data without asking for special user permission. We list the four side channels used in this study below.

1. Power consumption statistics.

   We computed a phone's power consumption by reading two files: `current_now` and `voltage_now` in the `/sys/class/power_supply/battery` directory. We computed power as `current_now` multiplied by `voltage_now.` The sampling interval was ten seconds. A higher sampling rate led to more fine-grained data, and therefore higher accuracy; however, we found that by increasing the sampling rate, the monitoring application itself may increase energy consumption and interfere with the results.

2. Audio and screen status.

   In the Android OS, AudioManager provides access to volume and ringer mode control. It can be used to get a phone's audio status. We monitored this feature by calling system API: `getMode()` and `isMusicAlive()` once every ten seconds. `getMode()` returns 2 if there is a phone call. `isMusicAlive()` returns True if background audio is present, False otherwise. Background audio includes video soundtracks, local or online music, and game sound effects. Considering that most games on the market come with Background Music (BGM) and sound, if the audio status returned True, we made the assumption that the user was watching a video, listening to music, or playing a game. Our experiment showed that even if the user switches their device into the mute state, `isMusicAlive()` will still return a 1; therefore, our approach's effectiveness was assured. We monitored screen status by calling `PowerManager.isScreenOn()`. It returns

True if the screen is on, which indicates the user is interacting with the phone, otherwise it returns False. For convenience, we refer to the latter as audio and screen.

3. Incoming internet traffic.

   Incoming Internet traffic is the amount of data downloaded to the device while connecting to Internet. We choose incoming Internet traffic as a side channel because users use their device to download content from the Internet more frequently than uploading. The user activities we mentioned before, such as watching videos and browsing the Internet, are activities that favored downloading over uploading.

   We parsed the file `/sys/class/net/wlan0/statistics/rx_packets` once every ten seconds to get Incoming Internet traffic. We recorded the number of received Internet packets. After observation, we classified the rate of traffic flow into three categories: (a) video scenarios, which provided the largest amount of traffic, and may come from media streaming; (b) Internet and game scenarios which had the second highest traffic, and resulted from webpage browsing, advertisement loading, and communicating with Google Play; and (c) Music, Idle and Phone Calls provided the smallest amount of incoming traffic. For convenience, we refer to incoming Internet traffic as traffic.

4. Shared virtual memory.

   Users can switch from one application interface to another when interacting with smartphones. For an example, a user may want to browse the Internet first, then close the browser to play a game, or open a music player. In this thesis, we define this switching between applications as a scenario transition. When a scenario transition happens, Android utilizes several system components to display the new user interface on the screen. This process causes a change in the shared virtual memory of these processes. Therefore, we were able to use this side channel to detect scenario transition.

To elaborate this, we will first describe two mechanisms used by Android to draw layouts: `BufferQueue` and `SurfaceFlinger`. `BufferQueue` acts as a "bridge" by connecting elements that generate graphics and elements that display them; we called the former "producers" and the latter "consumers". All movement related to graphic data transitions in Android devices relies on `BufferQueue`. The producer requires a free buffer and fulfills it with graphics data, then returns it to the queue. The consumer then acquires the buffer and processes the graphics data contained in the buffer. When it is finished, the buffer is then returned to the queue. `SurfaceFlinger` is an Android system component responsible for drawing display images. It accepts graphics data from multiple sources, composites them, and sends them to hardware. `WindowsManager` is a system component that interacts directly with applications. When `SurfaceFlinger` receives a call from `WindowsManager` to draw a display, it creates a layer consisting of `BufferQueue`. For an application, a navigation bar can be a layer, while the main UI can be another.

Most devices refresh their display at a rate of 60 frames per second; in order to avoid tearing, the display sends a signal called VSYNC to the system to inform it that it is safe to update the contents. Upon receiving the VSYNC signal, `SurfaceFlinger` goes through its layer list to collect all buffers. Once it has finished, it calls Hardware Composer to proceed to the next step. `SurfaceFlinger` only wakes up when the screen display is different than before.

`Hardware Composer` was introduced in Android 3.0 and has been kept through all succeeding versions of Android; it functions as the bridge between software and hardware. With `Hardware Composer`, Android will render `BufferQueue` in the most hardware-efficient way. Android's display usually contains several layers. For an example, in an application interface, keyboard and navigation bars are overlapped layers. `Hardware Composer` renders each layer into different buffers and passes all buffers to the display hardware.

When a user switches from one application to another, or when the phone goes to an idle state, the memory mapping of the `SurfaceFlinger` process will change due to this activity. We monitored this change to determine the time of scenario switching.

This feature was originally leveraged by Chen, Qian and Mao, 2014 [7]. Memory allocation information of the Android system is stored in the file `/proc/[pid]/statm`. Each access to this file returns one line of data, which consists of total-program size, total physical-memory size, shared physical-memory size, text code, library, private virtual-memory size, and dirty pages; each field is in a column.

We calculated shared virtual-memory size by subtracting private virtual-memory size from total virtual-memory size. In a typical scenario switch process, such as when a user switches from Video to Internet, the video application is closed; the home screen is then brought to the foreground, and then the browser is opened. In this process, the shared virtual-memory size of `SurfaceFlinger` changes due to the display change, which requires a buffer load and release. We monitored this feature to detect when scenario transitions take place.

Privacy leakage requires an application to be running in the background. In a real world attack, this could be a module hidden in a seemingly benign application. The application collects data continuously when the device is on, and sends it to a remote server. We used an offline server to process the data; however, it is possible to do all processing on the phone being monitored. Our application only requires permission to use the Internet, which is very common for apps in general. Our application requires no prior knowledge of an intended victim, which makes a scaled attack possible.

3.3 Decision Tree

We used a decision tree as classifier. Safavian and Landgrebe, 1991 [27] provided a detailed explanation of how decision trees works and how they are built. Decision tree classifiers have been widely used in fields such as speech recognition and natural-language processing. The main advantage of a decision-tree classifier is its ability to break down a complex problem into several smaller problems. A testing case will navigate down the tree to reach a particular end node; this end node indicates the group that the test case belongs to.

A properly designed decision tree classifier should be able to classify as many test cases as possible. It should be able to generalize beyond training data to gain a higher accuracy rate. Also, the tree structure should not be too complex: it should be simple to build and use in order to solve a large decision problem. Decision tree classifiers should also be flexible. When more training data comes, the tree should be easy to update with more branches.

Decision tree classifiers can be built with four methods: top-down, bottom-up, hybrid, and growing-pruning. In this thesis, we used the top-down method to build the tree. There are three aspects to consider when using the top-down method to build a decision tree. The first one is how to split the path. Various research has been done in searching splitting rules. For example, Wu et al., 1975 [33] designed a pattern for remote-sensing applications. Since our use of decision trees is only for specific data and not for a general-data classifier, we design our own splitting rules according to our data. The second aspect to consider when using the top-down method is the end-node decision, which should be determined when building the tree. In this thesis, we have three side channels, which make up three levels of the decision tree. Therefore, leaf nodes in the last level naturally become end nodes. The third top-down method aspect needing consideration is end-node label assignment; we used a majority vote to assign labels.

Chapter 4

Classification Methodology

We built our approach using machine learning. We first collected training data and separated the data into several groups. We then analyzed each group of data and extracted signatures from them as profiles of each group. Then we used those profiles to classify the testing data.

4.1 Single Scenario Classification

We defined our single-scenario classification phase as a classification based on six basic scenarios: Video, Game, Internet, Music, Phone Call, and Idle. We also tested two-composite scenarios: Internet + Music and Internet + Phone Call. Our goal was to distinguish these scenarios by the four side channel features we described in Chapter 3.

We first tested our application by running it in the background on a smartphone, and we collected pertinent data from the device for further training and testing purposes later.

We collected the training data on power, audio, and traffic by parsing system files and calling various APIs. We repeated the data collection process once every ten seconds. Each "data clip" collection lasted for two minutes; therefore, we had twelve data points in each data clip. We collected twenty data clips for each scenario, for a total of 120 data clips. Each data clip consisted of three time series: audio status, incoming network traffic, and power consumption. The length of the data clip was normally twelve but the length was less in some data clips. A sample data clip is shown in Table 4.1. The first row is the power consumption in micro-watts, the second row shows network traffic in number of packets; and the third row is the audio statuses. Given a clip, C, we use $C^{(p)}$ to represent the time series of the power consumption. $C^{(n)}$ and $C^{(a)}$ are similarly defined. During the data collection, we disabled all unnecessary background applications and only left processes that are critical for Android's normal functioning when collecting data.

Table 4.1: A sample data clip of side channel data in Video scenario

Power(mW)	749	740	922	789	604	803	793	819	891	812	706	732
Network(number of packets)	382	789	1185	1552	1557	1893	2255	2600	2993	3003	3380	3848
Audio status(0,1,2)	1	1	1	1	1	1	1	1	1	1	1	1

Figures 4.1 and 4.2 show the averages of power consumption and the sum of incoming network traffic in two-minute clips for the six scenarios. We can see that power consumption of the six scenarios can be divided into two groups: the larger group consists of power consumption data from Video, Game, Internet, and Call

scenarios; the smaller group consists of Idle and Music scenarios. It was very difficult to separate the scenarios within the same group, so we decided to treat the power consumption as a discrete value of two: low and high. This matched with our intuition that loading video streams, drawing 3D graphics for games, sending requests, and loading webpages all consume a large amount of energy. Besides, all three scenarios require the screen background light to be on, which also consumes energy. By comparison, the other three scenarios are more battery-friendly.

Similar to power consumption, we divided incoming Internet traffic volume into three groups: the largest consisted of Video; the medium-sized group was Game and Internet; the smallest group was Music, Idle, and Call. We found that Video had the largest incoming Internet traffic, while Internet ranked second and was much smaller than Video. Music, Game, Phone Call, and Idle all had a significantly smaller volume of incoming Internet traffic. This is reasonable because downloading a video stream results in more incoming network traffic, whereas loading webpage is much lighter. Music, Game, Phone Call, and Idle had lower incoming network traffic volumes. For the same reasons as with power consumption, we treated the network traffic as a discrete value of three: low, medium, and high.

Figure 4.3 and 4.4 shows stability for using power consumption and incoming network traffic as attributes. Each line in Figure 4.3 and Figure 4.4 consists of ten data points. Each data point is the average value of power consumption data in a scenario over two minutes. Average-power consumption of the six scenarios was rather stable. For incoming network traffic, the result varied to some degree, but we could still easily divide six scenarios into three groups. Video was certainly the one

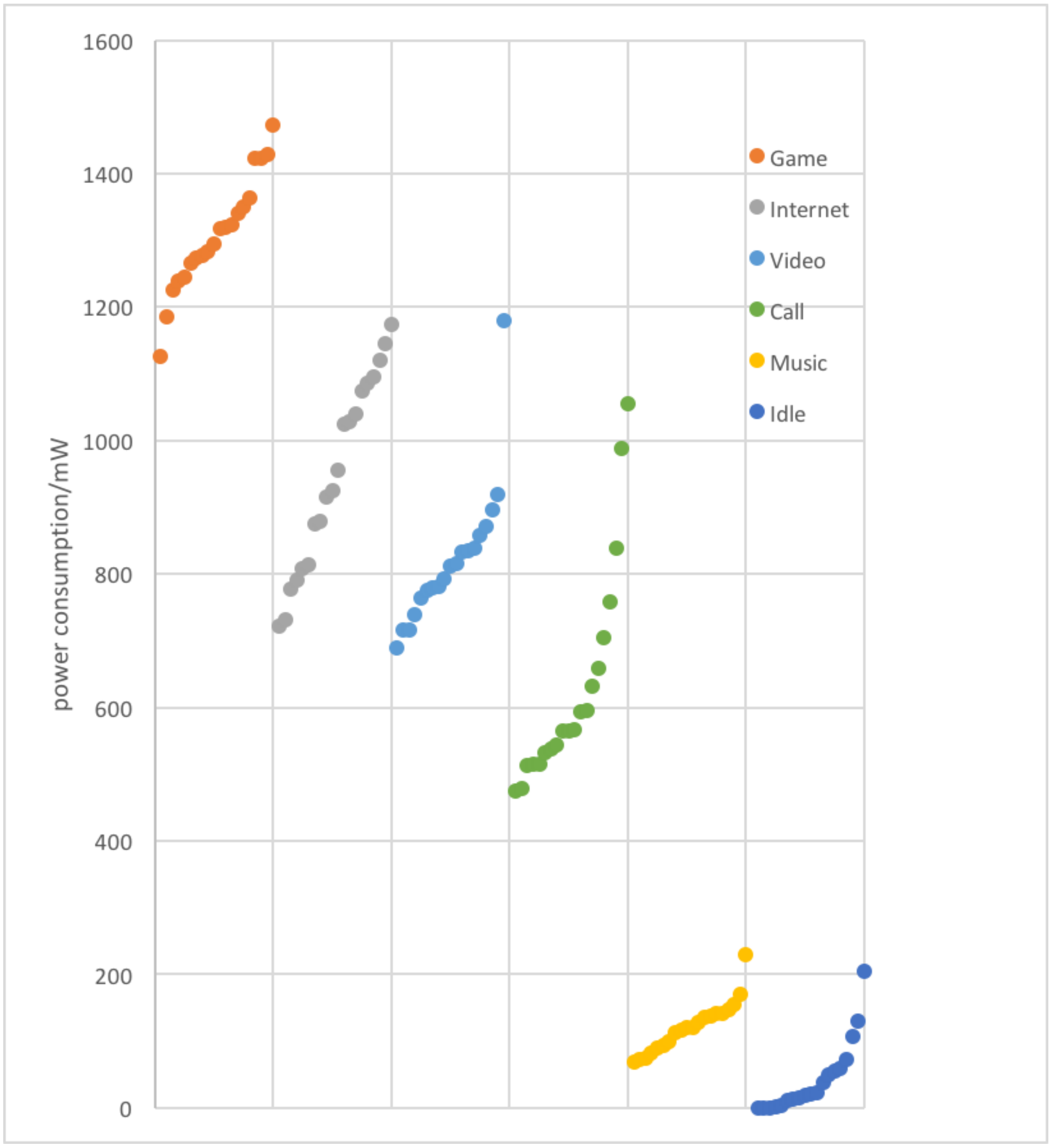

Figure 4.1: Power consumption examples of the six different scenarios.

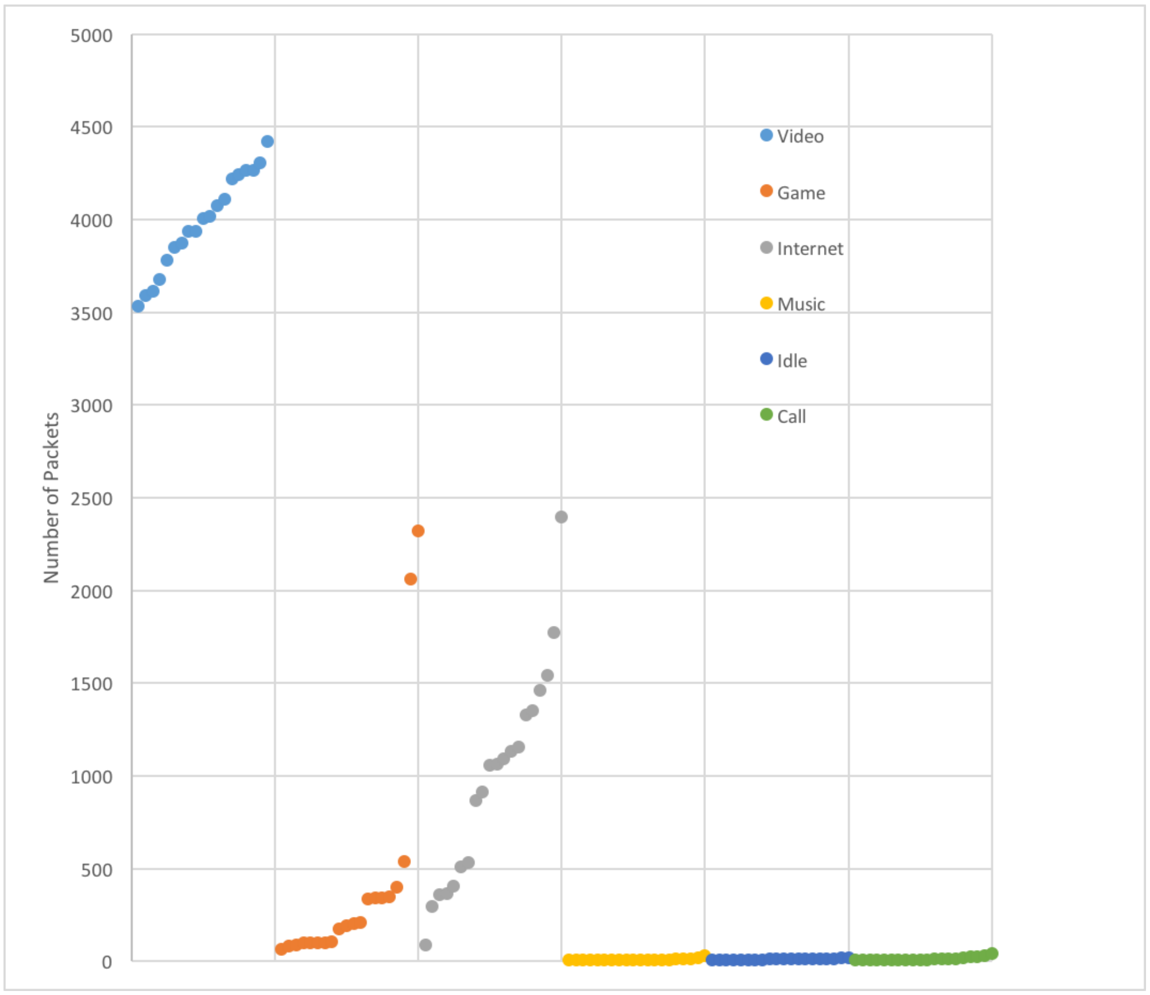

Figure 4.2: Sample Internet traffic load in various types of scenarios.

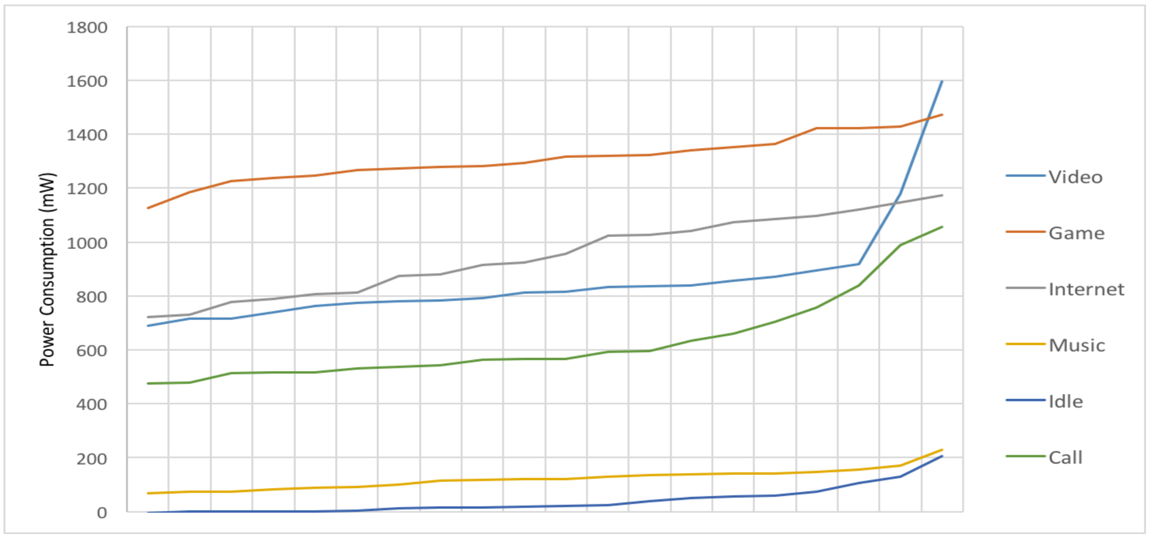

Figure 4.3: The changing trend of power consumption of six different scenarios.

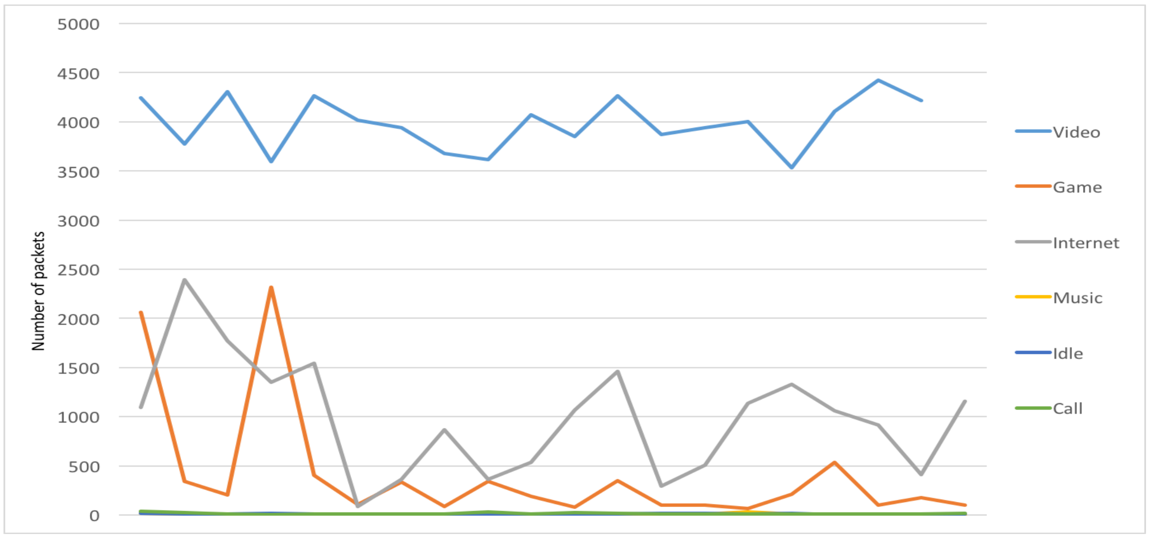

Figure 4.4: The changing trend of incoming network traffic of six different scenarios.

with most incoming network traffic, while Game and Internet came in second. Idle and Call used nearly zero network data. Notice that we did not need to separate six scenarios in this stage; we only needed to separate them into the three groups, which will be the intermediate result for our next step.

Audio data fell into three groups: "sound-on," "sound-off," and "calling." Video, Game, and Music all involve sound effects, so their audio status shows sound-on. Sound-off includes Internet and Idle. Since the system API `getMode()` reserves a special return value for Phone Call, it fell into a stand-alone group. We labeled the three groups as 0, 1, and 2. We extracted signatures from each group as profiles. Algorithm 1 details this approach as pseudo-code.

```
Algo. 1: Retrieving audio status data from OS
Input: None
Output: audio status of device
if getMode()==2
   return 2;
else
   return isMusicAlive()?1:0;
end
```

We called the API `getMode()` every ten seconds. For a two minutes long test case, we had approximately ten data points. Notice that the number of data points will not always be ten because of Android's repeating alarm mechanics [10]. In order to conserve battery, Android will not set an exact time-interval length between repeating alarms. If the API's return value is 2, a phone call is occurring; otherwise, it will return either 0 or 1, indicating the presence of background music. Our definition of background music includes playing songs and the background music of videos, as well as games.

As a result of the analysis of the data collected, we had eighteen ($2 \times 3 \times 3$) possible status combinations from the three attributes that we extracted. Our next task was to find profiles for each status value of the attributes. For example, when a test clip came in, we wanted to identify whether the power consumption fell under the discrete category of low or high. We used all the training data with "low" power consumptions to build the profile "low-power consumption"; we handled the other attribute statuses similarly.

Next, we describe how we build the profiles for the eight (=2+3+3) attribute statuses. Of the three attributes, two of them were time series, and the third was a discrete value. We first describe our algorithm for selecting profiles for the two time series. We measured the distance of the two data series by using the Dynamic Time Warping (DTW) algorithm [22]. The DTW algorithm has been successfully applied to temporal sequences of video, audio, and graphics data. Any linear sequence of data can be analyzed with DTW. By warping data points non-linearly in the temporal dimension, DTW calculates an optimal match between two series. It is independent of non-linear variations in time dimension. It can measure the similarity of any two linear series regardless of how quickly or slowly they change. The smaller the DTW value is, the more similar the two series are. Therefore, we were able to use the DTW distance to decide if two test cases belonged to the same scenario. We defined the distance between the $\mathrm{j_{th}}$ series and the $\mathrm{k_{th}}$ series as

$$dist(j,k) = \begin{cases} DTW(d_{\mathrm{j}}, d_{\mathrm{k}}), \text{for power usage and network traffic} \\ average_{\mathrm{i}}(d_{\mathrm{j}}[i] - d_{\mathrm{k}}[i]), \text{for audio attribute.} \end{cases}$$

We calculated DTW distances between all data series with the same attribute status. We then saved the index of the data series corresponding to the minimum distance value in a vector. We repeated this process until all data series with the same attribute status were processed. We chose three data series that appeared most frequently in that vector; these three selected data series represented profiles of that attribute status. Algorithm 2 presents this strategy in pseudo-code.

```
Algorithm 2: Attribute status profile building(e.g. low power consumption)

Input: All data series with the same attribute status

Output: A profile P representing the attribute status

for j = 1:n

   for each k = 1:n, calculate dist(j,k)

   M(j) = $\mathrm{argmin}_k$ (dist(j,:))

end

Choose 3 most frequently appeared time series as the profile for that attribute
```

By calculating the DTW distance between all training cases, we were able to choose specific cases that were the most similar to other cases in the same group. We wanted the selected profile cases we choose to have the smallest DTW distance within their group because we had to compare profile cases with testing cases waiting to be classified by calculating the DTW distance between them. Since groups we defined differed greatly from each other, we assumed that profile cases had a significantly larger DTW distance compared to test cases in other groups.

We used a decision tree to determine a corresponding scenario for a test data clip. We used profiles built with Algorithm 2 to build the tree. The root is labeled P for power consumption, and it has two splitting paths, which represent the two statuses of power consumption as described in Table 4.2. Nodes in the second level are labeled T for network traffic; each decision node has three splitting paths representing the three statuses of network traffic. The third level is labeled A for audio status; each decision node has three splitting paths, which represent the three values for audio status. The last level consists of end nodes.

Table 4.2: Attributes used for analysis and their Status Codes (SC)

Attributes	Data Sources	Status Codes	Value
Power Usage	current_now voltage_now	0	Low power usage
		1	High power usage
Internet Traffic	rx_packets	0	Low traffic
		1	Medium traffic
		2	High traffic
Audio on/off	getMode() isMusicAlive()	0	Audio OFF
		1	Audio ON
		2	Used by Phone call

For a given training data clip C, we have three components $C^{(p)}$, $C^{(n)}$, and $C^{(a)}$.

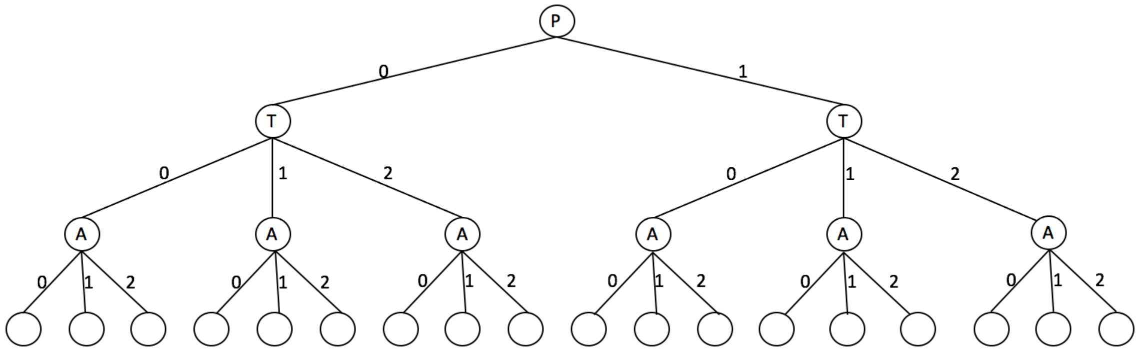

Figure 4.5: Decision tree before applying training data

Therefore, we had three data series for each data clip. $C^{(p)}$, which represents power consumption data, corresponds to the decision tree root. In the same manner, $C^{(n)}$ corresponds to the second level, and $C^{(a)}$ corresponds to the third level.

We use P_k to represent the three data series in a profile P. We define the distance of a data series d and a Profile P as

$$Dist(d, p) = min_k(dist(d, P_k))$$

Starting from the root of the decision tree, the algorithm navigates down the tree by choosing the attribute status with the smallest distance. Eventually, the algorithm reaches an end node; the data clip now falls into that node. Figure 4.5 shows the tree before labeling end nodes.

Each scenario had twenty data clips. We labeled the end node with the scenario fallen into it the most frequently. If a leaf node had multiple scenarios fall into it and the number of them was similar, we labeled it as Unable to Classify (UC). When a data clip fell into an end node, we stored the corresponding scenario of that data clip in the end node. We repeated this process for the 160 data clips from all eight scenarios. We also collected data for the two composite scenarios: Internet+Music and Internet+Call.

For testing data clips, we extracted data series from them and navigated down the tree in the same manner. The label of the end node it fell into was the scenario of that test case. Algorithm 3 models this logic.

```
Algorithm 3 Building a decision tree for classification
Input: All training data clips; Profiles for all attribute statuses
Output: A decision tree for data clip classification
Step 1: Initialization
    Build a tree as shown in Fig. 3.4 with one level for each data attribute
    and a blank label for the end nodes.
Step 2: Training
    For each training data clip C:
        Extract power consumption data $C^{(p)}$
        Calculate distance between $C^{(p)}$ and the two power consumption
        profiles (low and high)
        Travel down the tree based on the smaller of the distances.
        Network traffic $C^{(n)}$ and Audio status $C^{(a)}$ are processed in the same
        manner.
        When a end node is reached, store the scenario of this training data
        clip in this node.
    end
    Label each end node with the most frequently appeared scenario for that
    end node.
    If the number of multiple scenarios in an end node are similar, labeled it
as UC
```

4.2 Scenario Transition Detection

Our main goal was to detect not only scenarios, but also borders during scenario transition. We studied two types of transitions in user behavior. One was switching from an active scenario to Idle; in this case, the screen backlight goes off and the phone eventually falls into sleep. This kind of switching is easy to detect because Idle features differ significantly from those of active scenarios. The other type of switching is seamless scenario transitions during daily usage of a mobile device. For example, a user may open a browser to surf the Internet, and then close it and open a game application. Our goal was to detect the border between surfing the internet and starting the game. We focus on this type of seamless transition in the remainder this thesis.

First, we describe how we processed collected data as a stream. Then, we describe how we utilized the changing of screen status and the shared virtual memory size of `SurfaceFlinger` to detect borders.

We leveraged a sliding window to process data streams. Our client-end parsed a file every ten seconds and stored data in a device cache. It then sent data to our remote server every two minutes. This was to avoid sending data too frequently and draining the battery quickly. The server side stored all data it received in chronological order. We read data in a fixed period into the cache, then classified each feature into categories and used the decision tree to determine the corresponding scenario.

We refer to the classification of features and subsequent decision making as data processing. We defined the fixed period of time as the base window. Then, we continued reading in data column by column and concatenated these to the end of our previously read-in data. Each time we read a new column of data, we made decisions using previously stored data and new data, until the total-data length reached a certain limit, which we defined as the maximum window. We then discarded all data from the base window to the maximum window in the cache and moved the starting point to the end of the base window. We then started a new session, repeating the steps we described above. Newly made decisions overwrote previous ones. This gave us the chance to "correct" wrong decisions we had made. Accuracy rate was defined as the number of correctly classified segments divided by the total number of segments. Figure 4.6 shows this incremental method.

We found out that the values of the base window and the maximum window could affect the accuracy rate. Figures 4.7 and 4.8 show the accuracy rate changing with

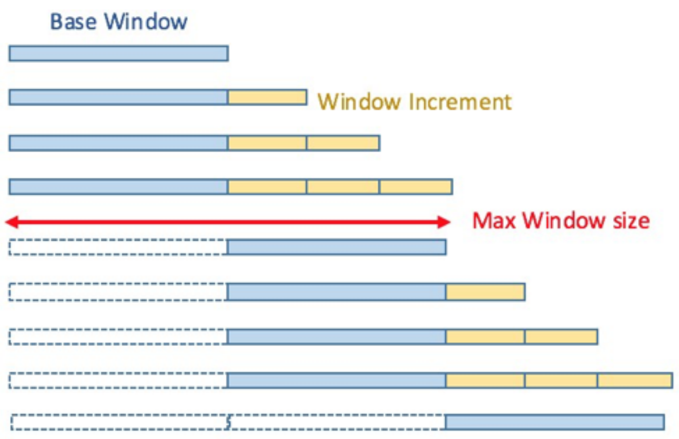

Figure 4.6: An illustration of using shifting base windows and increments for detection.

different base windows and maximum windows. The accuracy rate first rises, and then drops an increase in the base window. When the base window was small, we did not have enough data to perform classification. Therefore, the accuracy rate was low and would increase with the base-window size. However, after the base-window size reached a certain value, if we continued to increase it, we started to include data from another scenario, which lowered the accuracy rate.

We can see that the accuracy rate increased with the maximum window; this can also be explained with the amount of data. When we increased the size of the maximum window, we had more data to make a decision, which enhanced the accuracy rate.

To enhance our ability to detect a user's borders of seamless-scenario switching, we developed a multithread system. Our system consisted of a phone-basics thread, a screen-status thread, and a memory-tracking thread. The phone-basics thread held the processing of power, traffic, and audio. The screen-status thread

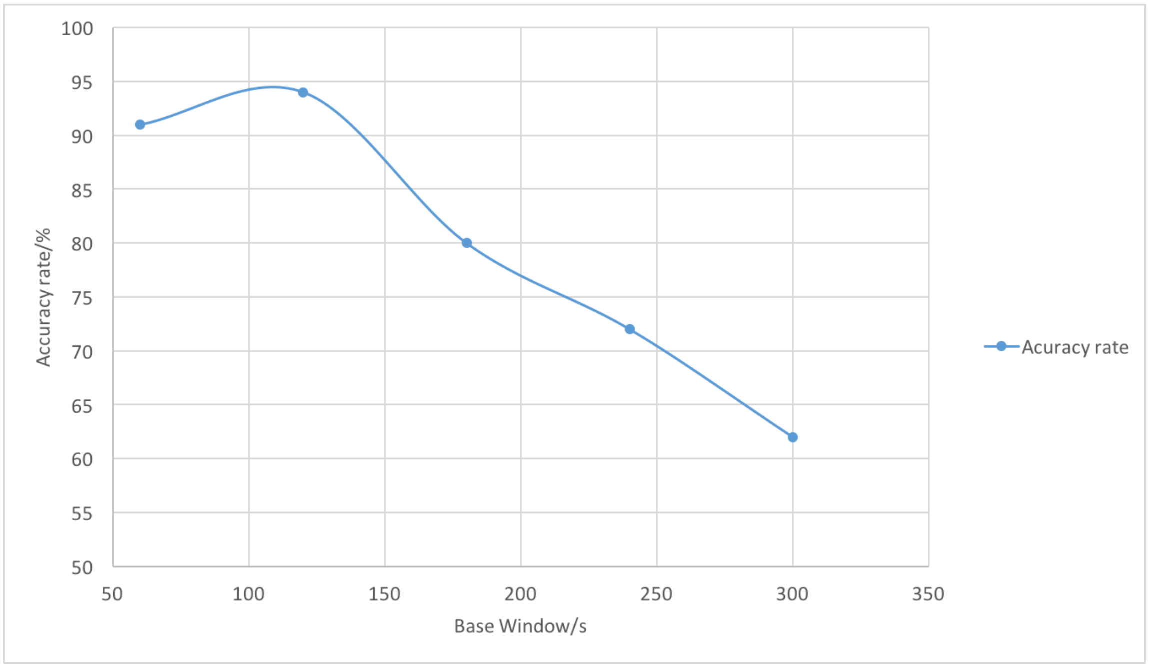

Figure 4.7: Relationship between accuracy rate and base-window size

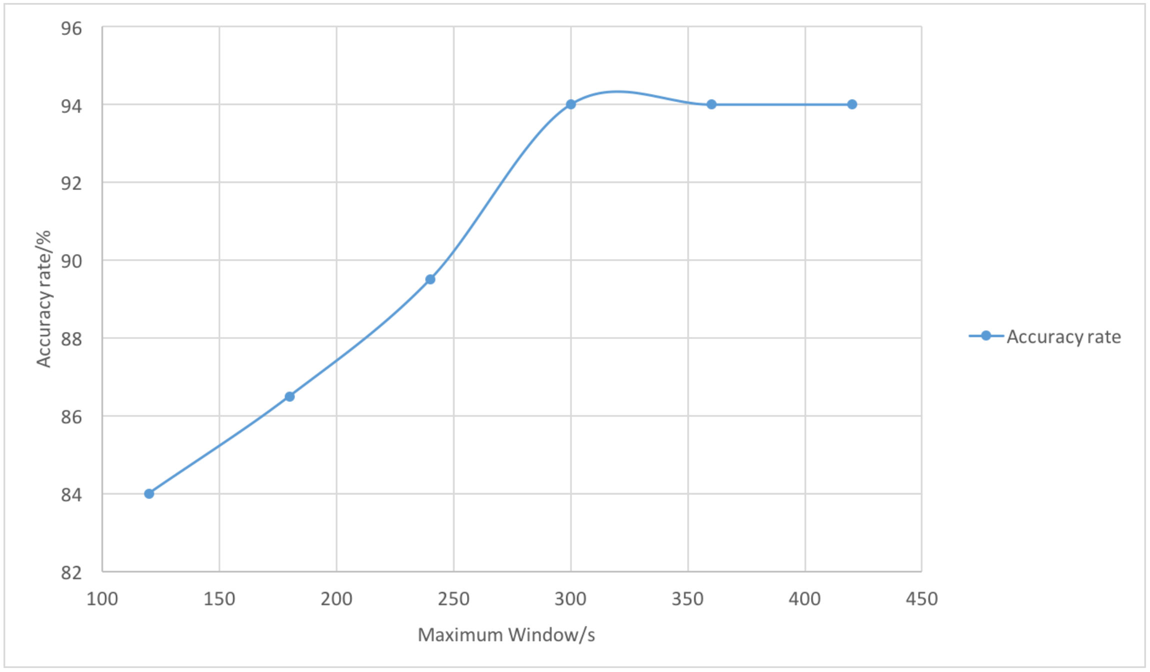

Figure 4.8: Relationship between accuracy rate and maximum-window size

monitored screen-status changes. If a change occurred, it sent an interrupt to the phone-basics thread. The memory-tracking thread monitored changes in the shared-virtual-memory size of a monitored process. If a peak was detected, it sent an interrupt to the phone-basics thread.

When the phone-basics thread received an interrupt, it performed data processing on data in the cache, and then started a new session. This process aimed to detect the borders of seamless-scenario switching. As we explained in Chapter 3, a change in the shared-virtual-memory size of SurfaceFlinger often indicates a switching between applications, and a screen-status change may also indicate a switching. The memory-tracking thread and screen-monitoring thread start working after the phone-basics thread has finished processing the base-window data, which is at the start of a session. This is because although "start a new session" effectively avoids a wrong judgment at a border, a short-data clip could also lead to a wrong judgment due to insufficient information.

Chapter 5

Experiment Design and Results

5.1 Building the Decision Tree

In this section, we describe the building of our decision tree using the algorithm introduced in Chapter 4. First, we selected three profiles for each category. Second, we used training data to compare profiles and navigate down the tree. Third, we recorded cases falling into each end nodes. After all training cases were processed, we selected the most frequently appearing scenario in the end node as the label.

We used a two-fold cross validation. We divided our data equally into two groups. We first used twenty data clips as training cases to select profiles and build the tree, and used the other data clips for testing. Then, we exchanged training cases and testing cases to rebuild the tree. The two trees had the same end nodes but showed little difference in data distribution. Figures 5.1 and 5.2 shows the training cases

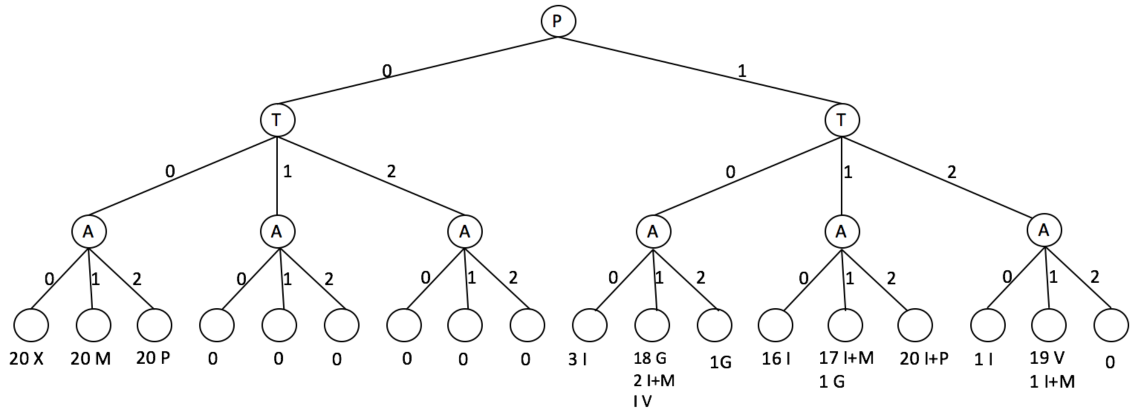

Figure 5.1: Training cases distribution for end nodes in first round of cross validation(P: Power, T: Traffic, A: Audio, X: Idle, M: Music, C: Phone Call, G: Game, I: Internet, V: Video)

distribution of end nodes in the tree. Table 5.1 shows the total number of training cases in the different scenarios falling into end nodes when building the tree. We used the same notation as in Figures 5.1 and 5.2. Figure 5.3 shows the tree after we processed all training data.

Table 5.1: Percentage of different scenarios' training cases falling into end nodes

	X	V	G	I	M	P	I+P	I+M	UC	Percentage(%)
(X)Idle	39								1	97.5
(V)ideo		38	1					1		95.0
(G)ame			37					2	1	92.5
(I)nternet				34					6	85.0
(M)usic					39				1	97.5
(P)hone						40				100.0
I+P							40			100.0
I+M		2	3					35		87.5

When building the tree, we saw that Video, Game, and Internet+Music were usually mixed together. This is reasonable, because all of those scenarios involve similar smartphone usage. First, all of the scenarios must have the screen on, which

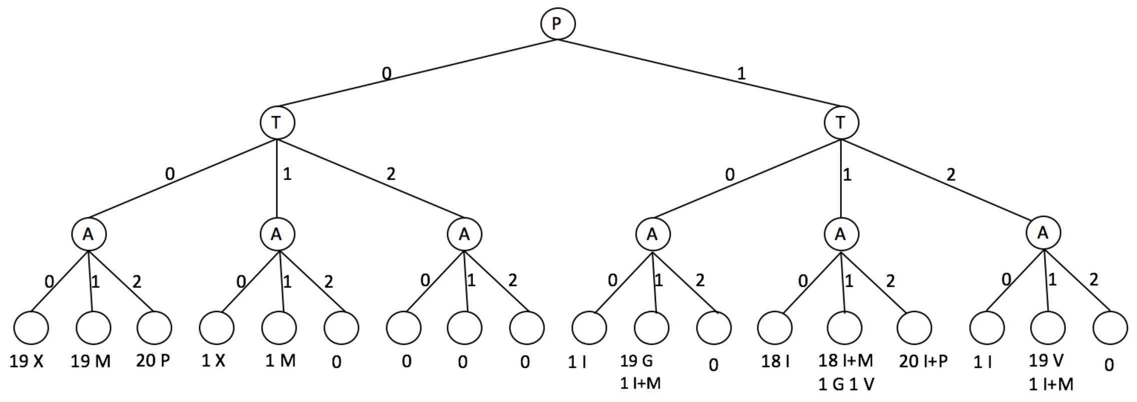

Figure 5.2: Training cases distribution for end nodes in second round cross validation(P: Power, T: Traffic, A: Audio, X: Idle, M: Music, C: Phone Call, G: Game, I: Internet, V: Video)

results in large power consumption. Second, all of the scenarios have background music. Remember that our definition of background music includes the background music of videos, games, and songs. For incoming Internet traffic, we noticed that some game applications had embedded advertisements, which caused an increase in incoming network traffic each time an advertisement is loaded. Also, some game applications have advertisements embedded internally, which therefore require online-data transmission.

Video length is another factor that impacts the results. When we watch videos on the Internet, the flash player preloads data. In a short Video scenario, the largest amount of incoming Internet traffic occurs at the beginning. It is possible for the video file to be so small that it loads completely at the beginning. Incoming Internet traffic during the remaining time of the scenario was lower than expected; this could cause a Video scenario to be wrongly classified as an offline Game.

Phone Call and Idle are the easiest because for Phone Call, we have a certain

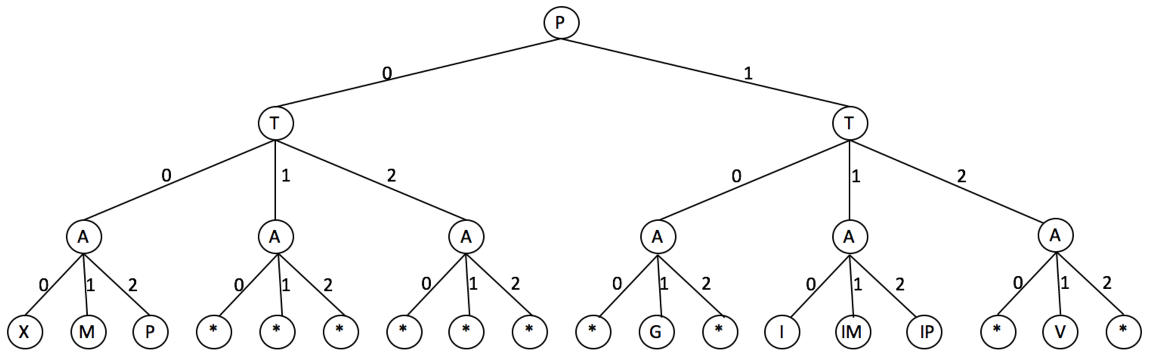

Figure 5.3: Decision Tree Used for Scenario Detection (P: Power, T: Traffic, A: Audio, X: Idle, M: Music, C: Phone Call, G: Game, I: Internet, V: Video, *: unable to classify)

value returned by system API; and for Idle, the phone usage is minimized, therefore all attributes will be close to zero.

5.2 Testing on Real Devices

Our experiments were carried out on a ZTE Z667T phone running Android 4.4.2 KitKat. In this version, the `getRunningTasks` API can be used to get foreground processes with permission `GET_TASKS`; however, we did not directly call this function as it was changed in Android 5.0. In the latest version (Marshmallow), a background application can only use this API to get information about itself, and can no longer be used to get foreground process information anymore. Android's sandbox mechanism also aims at preventing such conditions from happening, for safety reasons.

Despite that change, the side channels we used persist through different versions of the Android system, so we can safely assume that our method will remain effective with different versions of the Android OS. The CPU was a Qualcomm dual-core 1.2 GHz processor and the display was a 3.5" HVGA touch screen with 320x480 resolution. Available internal storage was 1.3 GB and the RAM capacity was 512 MB. We used YouTube and BiliBili Anime as video applications. Android's built-in music player was used to play music. The browser we used was Google Chrome. We used three game applications: Temple Run, Snake '97: Retro Phone Classic, and Plague, Inc.

5.2.1 Single Scenario Classification

Table 5.2 shows the results of accuracy rate of single-scenario classification. The result is an addition of two-fold cross validation. Each round has twenty testing cases. We found distinguishing Game and Internet scenarios to be the most difficult; three cases of Game scenarios went into Internet + Music while seven cases of Internet scenarios went into Unable to Classify. The possible cause of these results is that games, especially free ones available on the Android app market, usually contain advertisements. While a user is playing such a game, it connects to the Internet and downloads advertisements, which generates incoming traffic. A possible reason for the difficulty in distinguishing Internet scenario cases is that loading different webpages generates different amounts of incoming traffic, and some webpages (those with large images, for example) may cause an abnormally large amount of incoming traffic, making our system give inaccurate results.

Table 5.2: Distribution of test cases of six scenarios and average classification accuracy rates

	X	V	G	I	M	P	I+P	I+M	UC	Accuracy(%)
(X)Idle	38								2	95.0
(V)ideo		40								100.0
(G)ame		2	34					3	1	85.0
(I)nternet				33					7	82.5
(M)usic					40					100.0
(P)hone						40				100.0
I+P							40			100.0
I+M		3	3					34		85.0

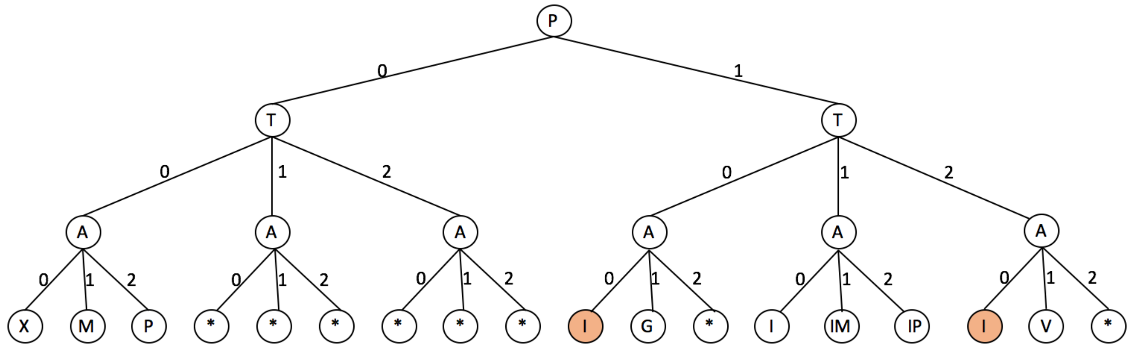

Figure 5.4: Altered Decision Tree Used for Scenario Detection (P: Power, T: Traffic, A: Audio, X: Idle, M: Music, C: Phone Call, G: Game, I: Internet, V: Video, *: unable to classify)

5.2.2 Change Labeling of Decision Tree

In Figures 5.1, 5.2, and 5.3, we can see Internet cases reaching three end nodes. We labeled the end node with the most cases: Internet, while the other two nodes were labeled UC (Unable to Classify). However, if we use a majority vote rule, we could also label the other two end nodes as Internet nodes. The altered decision tree is presented in Figure 5.4.

The main difference between the decision tree shown in Figure 5.3 and 5.4 is we labeled two more nodes as I (Internet). When we use this tree to classify test cases, we will get a 100% accuracy rate on the scenario Internet. That means all previous UC cases go to those two nodes. Below is the updated distribution table.

Table 5.3: Updated distribution table of test cases

	X	V	G	I	M	P	I+P	I+M	UC	Accuracy(%)
(X)Idle	38								2	95.0
(V)ideo		40								100.0
(G)ame		2	34					3	1	85.0
(I)nternet				40					0	100.0
(M)usic					40					100.0
(P)hone						40				100.0
I+P							40			100.0
I+M		3	3					34		85.0

5.2.3 Scenario Transition Detection

We then performed experiments for detecting seamless-scenario transitions. We used the format as scenario 1-scenario 2 to represent a scenario switch. We chose the five most difficult cases of all scenario-switching conditions: Video - Game, Video - Internet, Game - Internet, Video - Music, and Video - Internet with Music. Border of these scenario combinations are hard to detect for a variety of reasons. Video, Internet, and most conditions of Game scenarios often involve in similar amount of incoming network traffic, battery consumption, and audio activity. Video streams are usually the largest data consumers; however, due to the nondeterministic character of webpage loading and some large built-in advertisements in game applications, Internet and Game scenarios can also have large data consumption. The pairings comprised of the Video, Game, and Internet scenarios have the blurriest borders

Table 5.4: Border detection success rates and classification accuracy of 5 scenario transitions (with 5 test cases each)

Scenarios	Border Detection	Correct Classification
V-G	5/5	4/5
V-I	5/5	5/5
G-I	2/5	2/5
V-M	5/5	4/5
V-(I+M)	5/5	4/5
Accuracy(%)	88	80

between each scenario, and thus resulted in the greatest challenge for detecting.

The experimental results are presented in Table 5.3. We were able to detect the borders of the Video-Game, Video-Internet, Video-Music and Video-Internet + Music scenarios. For scenarios sharing the most similar features, such as Video and Game, accuracy was slightly lower than those scenarios not sharing as many features. For example, the Video and Music scenarios differed significantly in battery consumption, and the Internet scenario is usually occurred without an active audio, whereas Video usually did have this. By successfully determining the borders between two scenarios, we obtained a high-accuracy rate. When we found a border, we were able to cut the cached data stream into two parts and compare these with the training data we collected previously; this method eliminated any possible data-mixing effects near a border. As previously noted, border determination is enhanced by monitoring screen status and background-memory usage. However, we failed to detect a border between the Game and Internet scenarios, possibly due to advertisement loading and data exchange with Google Play; the Game and Internet scenarios also showed similar features in our measuring system.

5.3 Data Source Robustness Analysis

As we can see from results presented in Section 5.1 and 5.2, although we achieved reasonable accuracy rates, separating Video, Game, and Internet scenarios was still relatively difficult. We use three data sources: power consumption, incoming internet traffic, and audio status. Among these three, audio status is the most robust; these data is directly obtained from the system API, so we only had three possible values for it. Our premise is that the Android system is reliable; we conclude from this that we can be completely sure whether there is music on or not, or there is occurring or not.

Power consumption was the second robust-data source. We separated the scenarios into two groups: the first was those with low-power consumption (Music, Idle, Call); the other was those with high-power consumption (Video, Game, Internet). We believe the power consumption differences between scenarios is related to screen usage. If an user wants to watch a video, or play a game, or browse the Internet, they must wake up the screen. In contrast, Music and Idle are almost guaranteed to have lower-power consumption levels. Therefore, we were easily able to separate them using this data source.

Our relatively less robust data source was incoming Internet traffic. As we mentioned before, we found that the Video scenario did not always have the highest incoming traffic. For a Video scenario with short-time duration, we found incoming Internet traffic may peak at the start of the scenario, which will then be grouped into the large traffic group. It may also fall into the medium traffic group, or even the low traffic group; this may cause classification errors when trying to separate Video and Game scenarios. For Game scenario, even though we targeted offline games, it is still possible that the application contained online advertisements, as they are becoming rather frequent in free game apps. Advertisements will cause incoming Internet traffic to rise when loaded for the first time; some embedded advertisements are in video format, which makes separating them even harder. Our future work will focus on a more fine-grained analysis of incoming Internet traffic.

Chapter 6

Conclusion and Future Work

The Android system provides many APIs to support the functionalities of its apps; however, such APIs can become a source of privacy leakage, negatively affecting an unsuspecting smartphone user. In this work, we have demonstrated that even though one feature alone may seem harmless, multiple features combined together can leak information about a given smartphone and its user. If a data collection module is embedded in a normal application, mobile devices may be vulnerable during the downloading and installation processes. It is possible to have a remote server analyze collected data to obtain patterns of user behavior without knowing the identity of the device user. If the attacker does know more details about an intended victim, the amount of personal data compromised may be even higher, as demonstrated in Chapter 2. Our future work will focus on scalable attacks of this kind.

We analyzed the robustness of our data sources in Chapter 5, and our conclusion was that the incoming Internet traffic data source had the largest room for improvement in our means of analysis. We used the total number of package downloaded onto a device as the indication of incoming internet traffic. Our future work will focus more on determining more fine-grained data-usage trends according to time, and on how incoming Internet traffic changes between different scenarios.

We categorized complex user activities into six single scenarios and two composite scenarios; however, user activities in the real-world are far more complex than this. In the future, we will include more scenarios; for example, we will look into different game-playing activities, such as online games.

It is not enough to ban applications from accessing multiple features in one application, as it is possible to distribute a collection of these features over several applications to avoid raising safety concerns. When these apps are coordinated in such a manner, a user's privacy information may be seriously compromised.

There is also new concern about the stealthy phone calls made by coordinated smartphones, the so-called Telephony Denial-of-Service (TDoS) attacks[12] [35]. These malicious apps are able to make phone calls without alerting the operating system. We plan to see if some of our techniques are able to detect this type of attack behavior.

Bibliography

[1] Andriotis, P., T. Tryfonas, G. Oikonomou, and C. Yildiz (2013). A pilot study on the security of pattern screen-lock methods and soft side channel attacks. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, New York, NY, USA, pp. 1–6. ACM.

[2] Aviv, A. J., B. Sapp, M. Blaze, and J. M. Smith (2012). Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, New York, NY, USA, pp. 41–50. ACM.

[3] Bailey, M., E. Cooke, F. Jahanian, J. Nazario, and D. Watson (2005). The internet motion sensor: A distributed blackhole monitoring system. In *In Proceedings of Network and Distributed System Security Symposium (NDSS '05*, pp. 167–179.

[4] Becher, M., F. C. Freiling, J. Hoffmann, T. Holz, S. Uellenbeck, and C. Wolf (2011). Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, Washington, DC, USA, pp. 96–111. IEEE Computer Society.

[5] Bianchi, A., J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna (2015). What the app is that? deception and countermeasures in the android user interface. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, Washington, DC, USA, pp. 931–948. IEEE Computer Society.

[6] Büscher, A. and T. Holz (2012). Tracking ddos attacks: Insights into the business of disrupting the web. In *Presented as part of the 5th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, San Jose, CA. USENIX.

[7] Chen, Q. A., Z. Qian, and Z. M. Mao (2014). Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC '14, Berkeley, CA, USA, pp. 1037–1052. USENIX Association.

[8] Cisco (2014, October). The surging threat of telephony denial of service attacks. San Antonio, Texas.

[9] Czyz, J., M. Kallitsis, M. Gharaibeh, C. Papadopoulos, M. Bailey, and M. Karir (2014). Taming the 800 pound gorilla: The rise and decline of ntp ddos attacks. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, New York, NY, USA, pp. 435–448. ACM.

[10] Google (2014). Alarm manager. In *https://developer.android.com/reference/and -roid/app/AlarmManager.html.*

[11] Grace, M. C., W. Zhou, X. Jiang, and A.-R. Sadeghi (2012). Unsafe exposure

analysis of mobile in-app advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, New York, NY, USA, pp. 101–112. ACM.

[12] Guri, M., Y. Mirsky, and Y. Elovici (2016). 9-1-1 ddos: Threat, analysis and mitigation. *CoRR abs/1609.02353*.

[13] Hornyack, P., S. Han, J. Jung, S. Schechter, and D. Wetherall (2011). These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, New York, NY, USA, pp. 639–652. ACM.

[14] IDC (2015, October). Smartphone os market share, 2015 q2. In *http://www.idc.com/prodserv/smartphone-os-market-share.jsp*.

[15] Jana, S. and V. Shmatikov (2012). Memento: Learning secrets from process footprints. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, Washington, DC, USA, pp. 143–157. IEEE Computer Society.

[16] Karami, M., Y. Park, and D. McCoy (2016). Stress testing the booters: Understanding and undermining the business of ddos services. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, Republic and Canton of Geneva, Switzerland, pp. 1033–1043. International World Wide Web Conferences Steering Committee.

[17] Kührer, M., T. Hupperich, C. Rossow, and T. Holz (2014a). Exit from hell? reducing the impact of amplification ddos attacks. In *Proceedings of the 23rd*

USENIX Conference on Security Symposium, SEC '14, Berkeley, CA, USA, pp. 111–125. USENIX Association.

[18] Kührer, M., T. Hupperich, C. Rossow, and T. Holz (2014b). Hell of a handshake: Abusing tcp for reflective amplification ddos attacks. In *Proceedings of the 8th USENIX Conference on Offensive Technologies*, WOOT '14, Berkeley, CA, USA, pp. 4–4. USENIX Association.

[19] Memon, A. M. and A. Anwar (2015, November). Colluding apps: Tomorrow's mobile malware threat. *IEEE Security and Privacy 13*(6), 77–81.

[20] Michalevsky, Y., D. Boneh, and G. Nakibly (2014). Gyrophone: Recognizing speech from gyroscope signals. In *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA, pp. 1053–1067. USENIX Association.

[21] Michalevsky, Y., G. Nakibly, A. Schulman, and D. Boneh (2015). Powerspy: Location tracking using mobile device power analysis. *CoRR abs/1502.03182*.

[22] Müller, M. (2007). *Information Retrieval for Music and Motion.* Secaucus, NJ, USA: Springer-Verlag New York, Inc.

[23] Orthacker, C., P. Teufl, S. Kraxberger, G. Lackner, M. Gissing, A. Marsalek, J. Leibetseder, and O. Prevenhueber (2012). *Android Security Permissions – Can We Trust Them?*, pp. 40–51. Berlin, Heidelberg: Springer Berlin Heidelberg.

[24] Pathak, A., Y. C. Hu, and M. Zhang (2012). Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In *Proceedings*

of the 7th ACM European Conference on Computer Systems, EuroSys '12, New York, NY, USA, pp. 29–42. ACM.

[25] Paxson, V. (2001, July). An analysis of using reflectors for distributed denial-of-service attacks. *SIGCOMM Comput. Commun. Rev. 31*(3), 38–47.

[26] Pelechrinis, K., M. Iliofotou, and S. V. Krishnamurthy (2011, Second). Denial of service attacks in wireless networks: The case of jammers. *IEEE Communications Surveys Tutorials 13*(2), 245–257.

[27] Safavian, S. R. and D. Landgrebe (1991, May). A survey of decision tree classifier methodology. *IEEE Transactions on Systems, Man, and Cybernetics 21*(3), 660–674.

[28] Saravanan, P., S. Clarke, D. H. P. Chau, and H. Zha (2014). Latentgesture: Active user authentication through background touch analysis. In *Proceedings of the Second International Symposium of Chinese CHI*, Chinese CHI '14, New York, NY, USA, pp. 110–113. ACM.

[29] Seo, S.-H., A. Gupta, A. Mohamed Sallam, E. Bertino, and K. Yim (2014, February). Detecting mobile malware threats to homeland security through static analysis. *J. Netw. Comput. Appl. 38*, 43–53.

[30] Seo, S. H., D. G. Lee, and K. Yim (2012, July). Analysis on maliciousness for mobile applications. In *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pp. 126–129.

[31] Suarez-Tangil, G., J. E. Tapiador, P. Peris-Lopez, and J. Blasco (2014, March). Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Syst. Appl. 41*(4), 1104–1117.

[32] van Rijswijk-Deij, R., A. Sperotto, and A. Pras (2014). Dnssec and its potential for ddos attacks: A comprehensive measurement study. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, New York, NY, USA, pp. 449–460. ACM.

[33] Wu, C., D. Landgrebe, and P. Swain (1975). The decision tree approach to classification. Technical report, School Elec. Eng., Purdue Univ.

[34] Yan, L., Y. Guo, X. Chen, and H. Mei (2015). A study on power side channels on mobile devices. In *Proceedings of the 7th Asia-Pacific Symposium on Internetware*, Internetware '15, New York, NY, USA, pp. 30–38. ACM.

[35] Zetter, K. (2016, September). How america's 911 emergency response system can be hacked. *Washington Post*.

[36] Zhang, K. and X. Wang (2009). Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM '09, Berkeley, CA, USA, pp. 17–32. USENIX Association.

[37] Zhang, L., B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang (2010). Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the Eighth*

IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES/ISSS '10, New York, NY, USA, pp. 105–114. ACM.

[38] Zhou, X., S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt (2013). Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, New York, NY, USA, pp. 1017–1028. ACM.