Bachelor thesis
Computer Science

Radboud University

Using power analysis to differentiate between malicious repackaged apps and clean apps

Author:
Mick Koomen
S4468252

First supervisor/assessor:
Prof. Dr. L. (Lejla) Batina
`lejla@cs.ru.nl`

Second supervisor/assessor:
Dr. V. (Veelasha) Moonsamy
`v.moonsamy@uu.nl`

August 21, 2018

Abstract

Mobile malware can be detected in various ways, one of the ways is by looking at an app's power usage. Malware detection via power usage in mobile apps has been used in the past. So-called clean apps can have a significant difference compared to the same app repackaged with malware in terms of power usage. In this thesis we will show that the infection phase of certain malware can pay off in detecting malware by an app's power usage.

Contents

Chapter 1

Introduction

Mobile malware has been around ever since the introduction of Google's mobile operating system called Android. To prevent infections of malware, it is essential to be able to detect its presence on a device. Malware detection can be done in multiple ways, i.e. via static or dynamic detection. This thesis will be about dynamic detection, which means that detection will be done during the runtime of the malware. Dynamic behavior can be defined in multiple ways, e.g. the connections a program is making with other machines on the Internet or the API calls it invokes. These characteristics of a program can indicate potential infections.

Apart from network traffic or API calls, a program's behavior can also be indicated by the power it consumes. The behavior of the program includes the functions it executes, which in turn influence the power that is consumed on the machine. Expected power traces can be used as indicators to classify potential apps as clean or possibly malicious. Anomalies in the power trace can indicate that the app that is being run includes extra functionality. This is interesting with regards to malware. Android applications, later referred to as *apps*, can be disassembled and afterwards packed with extra code that can be executed when the app is repackaged again. Malware authors have been adding malicious functionality to clean apps in order for regular users to make it seem as if they are using the original app.

This thesis will be about the power traces of genuine apps and how they compare to the same apps repackaged with malware. Further research has been done on this subject [22][34][32][26][25], however, this thesis differs in when the power trace is taken. The power traces in this thesis were taken during the initialization phase of the malware. This entails the phase in which the malware infects the device. Running an app can cause this, but sometimes other events need to happen before the malware activates on a device. During activation the malware often uses more energy, since it has to set up itself on the system. It has come to light that certain repackaged malware do result in significant changes in the power trace. In this thesis we will look at different families of malware, of which some show significant differences compared to other malware that do not.

It should be made clear that further on in the thesis, we will refer to so-called "clean" apps as apps that have not been repackaged with malware. The malware that is described in this thesis is malware that has been repackaged in a *clean* app.

The thesis is structured as follows: The first chapter discusses malware in general and the malware that was used in this experiment. Then, the research, i.e. the methodology and the experiment, is discussed. Afterwards, the related work is discussed, and lastly we will conclude on the results of our experiment.

Chapter 2

Malware

This section will describe malware and the types of malware that were used. In this thesis we will be using samples from different families of malware.

Malware is a portmanteau of *mal*icious and soft*ware*, where malicious indicates the main functionality of the software. As the word malicious implies, malware is designed to provide malignant functionality. It can have multiple purposes, i.e. it is not only used to cause damage to systems, but can also be used to retrieve information from users who are using the system on which the malware is installed. Although malicious software has gotten a lot of attention recently due to Wannacry [15] and NotPetya [17], which were two malware families that resulted in damages on a large, international scale, it has been around since the 1970s. Malware is written with different motives, which include generating money, sabotaging systems, espionage, or hacktivism.

We have chosen to limit the scope of malware studied in this thesis to Android-based malware samples, because of Android's market share. Currently, it has the largest market share in mobile operating systems (77.3%) compared to other mobile operating systems [21]. iOS has the second-largest market share, with 19.4% of the market. Additionally, we opted for Android because during the initial research we found more power analysis apps for Android than for iOS.

Android has a central service that is used to distribute Android apps: Google Play. Apps can be downloaded to an Android device through the Play Store app, which is often pre-installed on Android devices. The Play Store filters apps based on compatibility with user's devices. The filter is based on different requirements of an app, which can be stated in the manifest file of an Android app. This file contains metadata about the APK file, such as the required API version, whether a camera is required, etc. This filter ensures that the apps that are donwloaded on a device can in fact be used on said device. Another benefit from the centralised system is that it allows Google to review apps released on Google Play. Although it does provide some form of mitigation for users downloading malicious apps, it is not foolproof. There have been multiple cases where malware was discovered on Google Play [18][19].

Besides the Play Store, there are also third-party app stores where users are able to download apps. This comes with added risk. Users are more likely to download malware. The Play Store provides a much larger audience for apps, which means that there is a higher chance of other users encountering the malware and alerting people of it. However, third-party app stores do attract users, since they can offer apps with free functionality or apps that are not available on Google Play. For instance, they could offer a premium game that does not require any payment.

2.1 Detection methods

The detection of malware can be done statically or dynamically [33]. The former does not execute the program and uses signatures to detect malware. Dynamic detection looks at the behavior of the malware during run-time.

Signatures that are looked at during static analysis are, for example, the calculated hash of the program or the strings it contains. Another static property in Android applications is the permissions that the app requests to be granted. The permissions of an app indicate what actions it may perform on the Android device. Static analysis can provide a one-sided picture of an executable and can be mitigated easily for an attacker. One may notice that the strings in an application could be obfuscated, or other malicious parts of the code can be decrypted during run-time and are therefore overlooked during static analysis. To spot these kind of obfuscated signatures of a program, dynamic analysis is needed.

Looking at the behavior of a program reveals more information about the program's intentions. Moreover, as has been stated in the introduction, this thesis is about a form of dynamic analysis of malware, namely the power consumption of an app during run-time. Different instructions lead to different power consumptions. For example, a calculator that calculates numbers would only need the CPU to execute instructions, while an app that communicates via the web would need to download and upload data. Downloading and uploading would require the app to use Wi-Fi or 4G - something that consumes extra energy.

2.2 Propagation

Malware can propagate itself via multiple ways. Mobile malware propagation can be described in three main methods. This thesis will look at malware that uses repackaging as propagation method.

As has been shortly stated in the introduction: a repackaged app is an application where the original app has been injected with extra functionality. With regards to malware, this additional functionality is malicious. The app often keeps its original appearance, in order to make it appear as the original app. This makes it more likely for an unsuspecting user to install this repackaged app and associated malicious functionality.

Furthermore malware can use so-called update attacks. This is another way to make it harder to be detected via static analysis. The app will include an extra update function [36]. Upon first inspection of the app, no malicious behavior will be detected, only after the user is prompted to "update" the app, the malicious functionality is added to the device.

The last method of propagation is a drive-by attack. This is a method where the user is visiting a website which hosts malicious code that downloads a .apk file unbeknownst to the user. This happened on a forum which served malicious ads. The ads would download malware when someone visited a page that hosted the ad [16].

2.3 Malware families

This subsection will describe the malware families of which we created power traces. Malware family names differ across AV vendors, however, we've adopted the names that were used in the research papers that described the malware data sets [36][31].

2.3.1 ADRD

The book *Android malware and analysis* [30] describes a lot of malware, among which ADRD. Its main purpose seemed to be increasing site rankings of Chinese websites on Baidu, a Chinese search engine. Baidu provides a service [28] where you can put a Baidu search box on a website and receive a share of the revenue generated from clicks on advertisements that would be received because of the embedded search bar. Increases in site ranking would increase the number of visits of a website, and the Baidu lookups would increase the share the Baidu affiliate would receive. It is important to note that the service that is started will only initiate contact with the command-and-control (C2) server after 6 hours have passed since the last contact.

2.3.2 BaseBridge

BaseBridge is malware which uses an update attack. When a user starts the application, it shows a prompt, asking the user to update the application [30]. After the "update", which installs a trojan, a reboot of the device is required. After the reboot, the trojan started different services, and tries to exploit a privilege escalation vulnerability. After privilege escalation, it will install the final payload, which would contain functionality, among which sending and deleting SMS messages, and performing phone calls [35].

2.3.3 Beanbot

This family is an SMS trojan, which communicates with a command-and-control server to retrieve premium numbers. After receiving the phone numbers, it will try to send text messages to these premium numbers in order to generate money [12].

2.3.4 DroidKungFu

This malware family has multiple variations. Upon initial execution it will try to execute two exploits to gain root privileges [11]. After gaining root, it decrypts an APK file and tries to install it. The installed .apk file turns the Android device into a bot by waiting for instructions from command-and-control servers.

2.3.5 Fakeupdates

Fakeupdates is a trojan, which upon execution of the app starts a service in the background which is able to retrieve .apk files from remote servers by informing command-and-control servers. Downloaded payloads will be presented to the user by the help of a fake update message. [10].

2.3.6 Gorpo

Gorpo is a malware family that is able to elevate user privileges. After this has been achieved, it will install an app in the /system/app folder, which is the Fadeb malware. Fadeb is able to download and run apps. The two malware families have said to be collaborating with each other [14]. We expect to see an increase in power consumption during the installation of the Fadeb malware. Gorpo, also called RealShell, uses a high amount of obfuscation, and builds an APK file from files located in the assets folder of the repackaged APK file.

2.3.7 Kemoge

Kemoge is a type of adware. During initialisation, it decrypts a file called bg.mp4. The file is a DES encrypted, password protected ZIP file, which will be decrypted and unzipped [13]. The malware contains a couple of root exploits, and will only communicate with command-and-control servers upon first launch or after 24 hours of no communication.

2.3.8 Pjapps

Pjapps is a trojan, which adds the infected device to a botnet [30]. A botnet is a network of bots, which are infected nodes that can act on behalf of someone who controls the bot. This is often done via a command-and-control server, which communicates with the infected machines. Pjapps is able to install other applications, send and block text messages, and visit websites.

Chapter 3

Research

3.1 Methodology

In this chapter the methodology of the experiment will be discussed. This entails both the preparation and the experiment itself. Lastly, we will discuss and compare the power traces that were the result from the experiment.

3.1.1 Preparation

In this section the methodology of the preparations that have to be taken will be described. The first part describes the way data sets can be gathered, while the second part discusses the power profiling app that will be used to measure power consumption.

Data set

To perform our analysis we first have to find samples of Android malware. These samples are .apk files, which are installed as applications on an Android device. In case of repackaged Android malware, the sample is the file in which the malware is repackaged. There are several ways to receive these samples. Websites can collect malware samples, e.g. the site VirusTotal [7], a website where people can upload hashes or files. VirusTotal then shows the amount of antivirus vendors that flag the file or hash as malicious or not. It is possible to retrieve malicious Android apps from such websites. Another way of retrieving malicious .apk files is by using data sets of researchers who have been doing research on a certain data set. The latter has been used in this thesis, as will be discussed further on.

In this thesis the focus is on the detection of malware via power traces, however, this means we are comparing power traces. By grabbing malicious .apk files from the web or from a data set, we retrieve one half of the comparison, the second half will be the corresponding clean .apk files. These will be downloaded from websites such as `https://www.apkmirror.com/`, https://www.apkmonk.com/, and https://www.apkshub.com/. These are often websites that host a lot of different Android apps. It should be noted that the downloaded apps could contain malware, therefore before assuming it was the original version of the app, the downloaded apps were uploaded to VirusTotal and were only used if most or all of the detection engines flagged it as not malicious.

Different malware families will have different purposes, e.g. Kemoge has a different purpose compared to Pjapps. Malware variants are about different versions of malware from the same family. This is the case with the DroidKungFu family, which has multiple variants. The data set that will be used contains malware from different families, this is done in order to see if certain behavior can be generalized among families. However, it can also be possible that certain characteristics only apply for one family and not for others. By using multiple families in our data set the differences and similarities between them can be shown. Of each family multiple samples are tested, however, only if multiple clean original apps could have been found for that family.

Trepn Power Profiler

To measure the power consumption of an app we use the Android app called Trepn Power Profiler. Trepn Profiler is a project by Qualcomm Technologies [6]. Trepn Profiler monitors the power usage of an Android device. Although it measures power usage, the site does state that unsupported devices could cause inaccurate reportings of power usage. There is a list of supported mobile devices [8] which shows Android devices that are supported. In the experiment, the LG Nexus 5 will be used, a device which is on the list of supported devices.

The reason we chose for Trepn profiler is based on its accuracy and the format in which the power trace is saved. In a paper that compares power monitor software [24], Trepn Power Profiler was stated to have an accuracy of 99%. In other papers researchers have used PowerTutor [5]. However the reported accuracy was said to be lower, namely 97.5%, which is partly due to PowerTutor not considering the GPU's power consumption. It must be stated, though, PowerTutor is able to trace power consumption per application running on the device, while Trepn Power Profiler is not.

3.1.2 Experiment

The goal of the experiments is to find whether there is a significant difference in the power analysis of clean apps versus that of their corresponding repackaged malicious app. To achieve this, there should be a correlation between the power analysis of an app and the activity of it. During the experiment multiple apps will be run on an Android device, the Nexus 5, and during the execution of the app its power consumption will be noted.

To test if there is a difference in an app's power usage, first, we have to decide whether it is better to use an input file, or to let the app run for a certain amount of time without user interaction. We have chosen to not give input to the apps during the monitoring of the device. This is done, because the malware does not trigger based on user input, but on events that happen on the phone, e.g. when there is a change in connectivity of the network. We try to trigger each malware by looking at what initiates the malware, and replicating this during the power analysis.

To perform a more reliable power analysis, a backup of the full phone state has to be made. This will result in both the clean and the repackaged app running from the same state. This is done by rooting the phone and then using custom recovery to create images of the phone state.

When the base phone state is saved, the experiment can start. For each app executed, we first revert the phone state to the base state. Then we will start the app and measure its power trace.

3.2 Preparation

In this section the actual preparations that have been made will be discussed. The first part elaborates on the data set that has been used. Then we will talk about how the phone state is saved. The last part will describe how we triggered the malware.

3.2.1 Data set

As has been stated in the methodology; the malware samples in this thesis were collected by using data sets from previous researchers that were doing research on malware.

The samples that have been used in this experiment are listed below. First there is the table of clean samples, then comes the table that contains information about the malicious samples.

3.2.2 Clean .apk samples

Package name	Malware family	Hash	Version	Size
com.tobyyaa.superbattery	Pjapps	df67b685c590acb1e9fa02f4090544b0481c47400ab59dcbad25c0ee17cce8bd	2.3.9	567K
uk.co.neilandtheresa.Vignette	Pjapps	d262d85c937fb96565ba1f25ab0c6520c82af5bb063a2e635dd1a698763aa5ee	2012.02.27	173K
com.nemo.vidmate	Gorpo	50225f3c729847bdf09f57b8907d2cf7f2c9d0df8a1a00e147808e45297724a6	2.11	3.3M
com.mobile.indiapp	Gorpo	09f7295ac1f6acb0536d6dd21ce3d03efc6bc57427b596645fece94e2f96c4b6	2.1.7.0	1.5M
com.youba.translation	Gorpo	99bcbfb1e5f872e75c72c8a5c27d502ac0f890370566ffef808def3737f9e9a0	1	168K
com.forthblues.pool	FakeUpdates	41a19985ced71d8eb49c8f7d35fffc9d5d92e881908a91e7cad788003d2b6588	2	2.3M
cc.toasha.beautify.easylocker	Kemoge	a4d0325cf6c326c9e20fff79a8223f781be139be04cff49e9389b19c195ea394	1.0.0	3.3M
com.leo.appmaster	Kemoge	c50c66e73f8c29c97ef86c084ec41f98e4df705f723e453904b7649387213ddc	1.6	1.8M
com.noisysounds	ADRD	2d327448c6c1381c3e87de26f41e70a52e10eef8c0d4dbd5f0c50dab88d3d4b6	2.0.0	1.1M
com.iPhand.FirstAid	BeanBot	7063915c1967170629e0a2ef09f1d225b252fb05009a73b80f6cde8a8d42bd90	1.0.0	557K
net.lucky.star.mrtm	BaseBridge	ede670a1fc42f1e49ea2d1bd3d0bea60c05d6048b5d17b7dc9e5a34004e56c05	1.95	438K
com.mechanics.engine	BaseBridge	7391004af85ca37f664e35c739b9191641a36a5fc8c234929b3c23c739f2dedc	2.0.1	2.3M
com.kayac.bm11.recoroid	BaseBridge	db242966b200498e90f65b49c786cd5e679487f4fa4293f6930b73fb4de62eee	2.4.1.7	243K
com.edwardkim.android.screenshotitfull	BaseBridge	b33f791db2275fd5ba353cef86be37b76b33e3f3d2cf4bbb8305271eaf5e3ebe	3.35	745K
com.bottleworks.dailymoney	DroigKungFu2	ce4884f48dac2a46de0d3efa78bb06574dc80b1357a24ac1deae1770a8432759	0.9.8-121107-freshly	660K
com.replica.replicaisland	DroigKungFu3	e41a08d84682e8aaa962faf22cc6ce8949426ad8c9c8e84d56c05899b23532dc	1.4	5.0M
org.openintents.filemanager	DroigKungFu3	466b6bbd5f62b75e15c1bfa317f881cfc80a0956ed6118787de4d07982b31f72	1.2	724K
com.glu.android.dinercn	DroidKungFu4	6956b037c0302273648704aa70fcd4104fe555c84d712b1f551b98b519c86f0d	1.4.1	3.7M

3.2.3 Malicious .apk samples

Package name	Malware family	Hash	Version	Size
com.tobyyaa.superbattery	Pjapps	1fad7afd7ccae3c7f70b11a342e52d4bae2a2f7a	1.6.8	239K
uk.co.neilandtheresa.Vignette	Pjapps	dd830e1a37a73816f138cb7dca076224f39e2ee6	2010-10-19	170K
com.nemo.vidmate	Gorpo	5872834ea2e939895d1bceeb08713ce0	2.31	5.3M
com.mobile.indiapp	Gorpo	53465cf9e999d3915ae5b9fe794e8fc8	2.1.7.1	2.1M
com.youba.translation	Gorpo	2f6e5617182fab57f6c162f9d89df8d3	1	1.2M
com.forthblues.pool	FakeUpdates	d386cfc6d17392ecf72e7b7a548dc65c	1.3	1.7M
cc.toasha.beautify.easylocker	Kemoge	2701de69ea6b57bbc827830660711ea2	1.0.0	3.4M
com.leo.appmaster	Kemoge	f1a16304e427b7f8657de8c3dfb1d33f	1.5	3.1M
com.noisysounds	ADRD	b999d381d96cc3b40edce7048543f8a7a3f0e79e	2.0.0	1.2M
com.iPhand.FirstAid	BeanBot	e8831cd2ec2e091cc3291821697851b1c65125d4	1.0.0	568K
net.lucky.star.mrtm	BaseBridge	f8c6d33e8dbd2172654bae104a484fcd80cf22ba	1.95	1.1M
com.mechanics.engine	BaseBridge	c0a74df6816864a66cdadfdd09114ee36f1622f7	2.0.0	2.6M
com.kayac.bm11.recoroid	BaseBridge	24d3a18e042c35eea3f95da16c7697ac15fe223b	2.4.1.7	874K
com.edwardkim.android.screenshotitfull	BaseBridge	f1079462762cc7bec822e97a3f1bfcfd5fa313cd	1.991	1.2M
com.bottleworks.dailymoney	DroigKungFu2	46ec6903a6c621c7b03d36224a79873b135ef2de	0.9.6-0320-freshly	602K
com.replica.replicaisland	DroigKungFu3	44637a506466199d4707fd75b03194f13174ddb4	1.3	5.5M
org.openintents.filemanager	DroigKungFu3	452d6acc582168089f8462bfc169b5c3e555ea2e	1.12	552K
com.glu.android.dinercn	DroidKungFu4	00b89bcb196a138f9f20a6853c41673a18a2575f	1.2.6	3.4M

Some families only have one sample to test, this is due to us not able to find the corresponding clean apps. To search for the clean samples, we used the following Google search queries:

- "PACKAGE_NAME" apk download
- "PACKAGE_NAME"

In the list above, PACKAGE_NAME is the name of the package retrieved from the malware sample. The package name was put in between quotes, to ensure Google only returned entries which contained the whole package name in its results.

Since we are comparing power traces, it is important that the apps that are being compared are almost the same. In the most ideal situation, the apps are exactly the same, except for the malicious code that has been added. In this situation the only significant power differences are likely to be caused by malicious code. We therefore tried to find original apps that had a version closest to the repackaged app. Unfortunately, most of the times this was not possible. The version numbers have been mentioned in the two tables.

3.2.4 Phone state

Saving the phone state of an Android smartphone can be done with the use of the built in backup service, which is provided by Android. However, this method only saves user created data, which resides in the data partition [27]. Another way to create a backup, is by creating a NANDroid backup. This is a logical copy of the whole internal memory [23]. To do this, the device has to be rooted. This could be done trivially with the use of a software toolkit. The Nexus smartphone series have a toolkit called the Nexus Root Toolkit. This is a tool which can automatically root a Nexus smartphone. It can also be used to install a custom recovery tool, named TWRP. Custom recovery is third-party software which replaces the stock recovery software on the Android device. With TWRP we can create the NANDroid backup.

It should be noted that TWRP does not restore the internal storage. Since apps can add extra data in the internal storage, we created a list of the folders that were present after the Trepn Profiler was installed and the backup was created. After each run the folders that were added to internal storage were removed.

3.2.5 Triggering malware

Malware is often triggered with a certain action the user or device makes. This differs between families. It is essential for this thesis that each malware triggers successfully in order to give a significant conclusion about the differences found in the comparison between the power traces.

To do this, we looked at what triggered the malware and create that same environment during the execution of the sample. When we looked at our malware samples, there were three ways the malware could be triggered. The user can be prompted to update their app, which will download or install the malware after the user complies. Secondly, the malicious APK file declares a receiver together with a service in the Android manifest file, where the receiver waits for a certain event and will then activate the malicious service. Thirdly, the malware has hooked a function that is called within the app. When this function is called, the malware will activate.

The manifest file of an APK file is a mandatory file that entails information about the app. This includes information such as the package name, the different components of the app, the permissions it needs, and other information, e.g. whether the app requires a camera. The different components of an app can be activities, services, receivers, or providers.

Services are pieces of code that run in the background, regardless of whether the app is running. This is ideal for malware writers, since this allows for persistence. This is because they can start a service with the help of a receiver, without requiring the app to be running.

The receiver is a component that is able to catch certain events that occur on the system, even when the app itself has not been started. In the manifest file it can be declared on what events the receiver will be woken up. The system events are wrapped in an *Intent* object and include, for example, `ACTION_BOOT_COMPLETED`. This is an intent that will be broadcasted whenever the device has booted up. Like the intent about the system that booted up, intents are also sent when a user has received an SMS or when the phone state changes.

In the samples used in this thesis, the malware often used receivers and services. When the receiver would catch a certain event, it would start a malicious service, which was often the only use of the receiver.

To look at the different things that trigger the malware, we used tools to retrieve the source code of the samples. We used three tools for the analysis;

- Androl4b [1];
- JD-GUI [4];
- dex2jar [2].

The initial analysis was done with the use of Androl4b, which is a virtual machine which has pre-installed tools that can be used to analyse APK files. Androl4b has a local web server running the Mobile Security Framework. This framework is capable of static and dynamic analysis of uploaded APK files. However, MobSF is not always able to decompile the Dalvik bytecode.

Dalvik bytecode is the format which is used by APK files to store the program instructions. This bytecode is saved in .dex files and stands for Dalvik EXecutable.

Dalvik bytecode can be decompiled to Java files in two ways. The first method decompiles the Dalvik bytecode to Java in one go. The second method takes one intermediate step. It first decompiles the .dex file to a Java archive, containing multiple files in Java bytecode. This step is done by the dex2jar tool. The second step is decompiling this .jar file back to a representational form of source code. It is called bytecode, since a processor cannot execute the instructions as it is. There needs to be a translation from bytecode to native code. The translation of bytecode to native machine code, that can be executed by the processor, is done by Android Runtime (ART).

If MobSF failed to decompile the Dalvik bytecode, we used dex2jar in combination with JD-gui to extract the source code of malware samples. Sometimes it was not possible to view the source code correctly. This can be caused by bytecode instructions that are not representable in Java source code. To tackle this, we viewed Smali code. Dalvik bytecode can be disassembled to Smali code, which shows the instructions the program executes. The only downside is that it is harder to read, however, it provides a more detailed representation of the app.

Sample information

After looking at the different malware, we wrote down which intents were able to trigger the malware. These intents would then be sent during runtime. The table below does not contain information for each sample, since the difference was minimal between different samples of the same family.

Some malware had certain checks it performed, such as DroidKungFu. This malware sets a timestamp when the service starts for the first time. Then, each time it starts, it checks if an hour (or half an hour) has passed. To mitigate this, we sent an intent which triggered the malware, then, we set the time a significant amount of time in the future, at last, we would trigger the malware again. Since in all cases of DroidKungFu the time will be set a significant amount in the future, we can regard the families' triggers as the same in the table below. The date of the device can be set with the following command *date mmddhhmmyyyy*, where the first two letters represent the month. To set the date, root is required.

Sample information		
Malware family	Trigger	Time of trigger
Pjapps	`android.intent.action.SIG_STR`	8-10 seconds
Gorpo	Launching the app	
FakeUpdates	Launching the app	
Kemoge	`android.intent.action.USER_PRESENT`	8-10 seconds
ADRD	`com.lz.myservicestart`	8-10 seconds
BeanBot	`android.intent.action.PHONE_STATE`	8-10 seconds
BaseBridge	Update prompt	Upon launch
DroidKungFu(2-4)	`android.intent.action.BATTERY_CHANGED_ACTION`	15-20 seconds

In some cases the malware triggered automatically after running the application, this was the case for Gorpo. Gorpo adds a function to the launcher activity that will thus be started upon launching the app. Then, the activity that is created will launch the dropper, which will retrieve two other .apk files.

Another family that automatically starts upon launching the app is FakeUpdates. This malware uses the same technique as Gorpo in triggering the malware.

For most malware that needed to be triggered we triggered them after 10 seconds, however, for DroidKungFu more steps were needed, therefore it triggered slightly later. One of the triggers of DroidKungFu was `android.intent.action.BATTERY_CHANGED_ACTION`, this intent is broadcasted relatively often, which meant that as soon as the date had been changed, the malware triggered.

After determining the triggers, we tested the samples on the phone with the use of Frida [3]. With this software it is possible to view a predetermined set of system calls that a program invokes. This allowed us to determine if the malware was triggered. Besides Frida, we also used Wireshark [9] to check if the sample communicated with a C2 server.

Because some of the triggers can also cause increased power consumption when a clean app is running, we have added the triggers for some of the clean apps as well. Namely, for Pjapps, Beanbot, DroidKungFu, ADRD, and Kemoge. The other apps prompted the user for action, or would activate automatically.

The next section will elaborate on the power traces that were obtained during the experiment.

3.3 Power traces

The Trepn Profiler stores the power traces in .csv format, which makes it easy to process. After making the power traces we first processed the data to extract the power data. The resulting traces are listed in Appendix A

A.1.

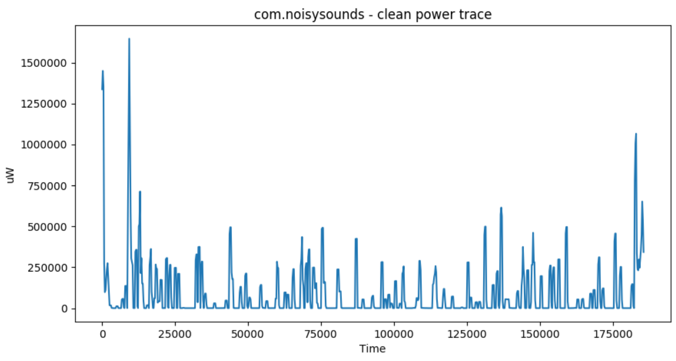
3.3.1 Comparing power traces

This subsection will discuss the differences found in the power traces. However, before we discuss the results, we will preface this by stating some facts about the power traces. Due to certain circumstances, such as differences in sample versions or slight differences in machine states, differences in power traces arise. To counter these environmental factors we set a time on which a certain malware would be triggered. Still, for the malware samples that triggered by themselves, this could not be accomplished. Because of the differences in versions, we cannot detect smaller differences in the power traces caused by activity from the malware.

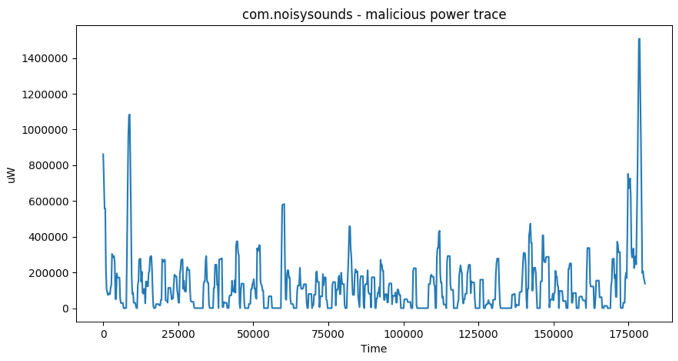
The sections below will show some of the power traces, however, not all of the traces will be shown. All power traces retrieved during the experiment are listed in the appendix A.1 which is located at the end of this document.

ADRD

ADRD upon infection sends a small amount of data to a C2 server, however, it does not do other things during initialisation, e.g. deobfuscation or installing files. When we compare the clean with the malicious power trace, we see no great differences. Both the clean and the malicious power trace show a spike around the 10 second mark, with the clean power trace showing a larger power consumption. When we compare the average power consumption from 8 seconds up to and including 10 seconds, we see that the clean and malicious power traces' averages are close to each other. The average of the clean power trace is 609768 microwatt, compared to 590353 microwatt for the malicious power trace. The clean power trace's average is 3.3% higher.



(a) Clean trace

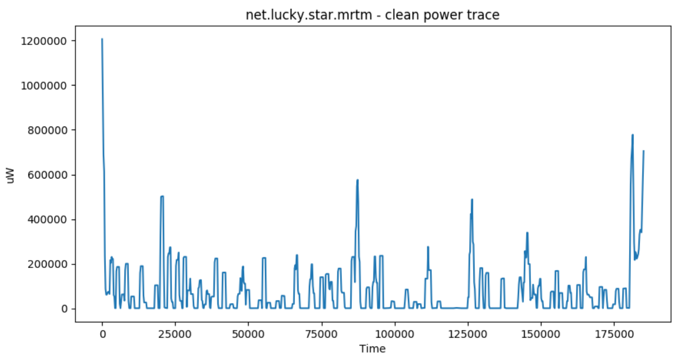


(b) Malicious trace

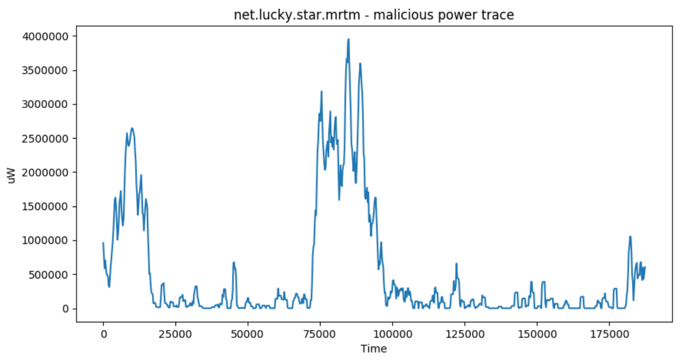
Figure 3.1: com.noisysounds

BaseBridge

BaseBridge is one of the families that shows a significant change in power consumption. As can be seen in the below graphs, around the 10 second mark there is an increase in power usage.



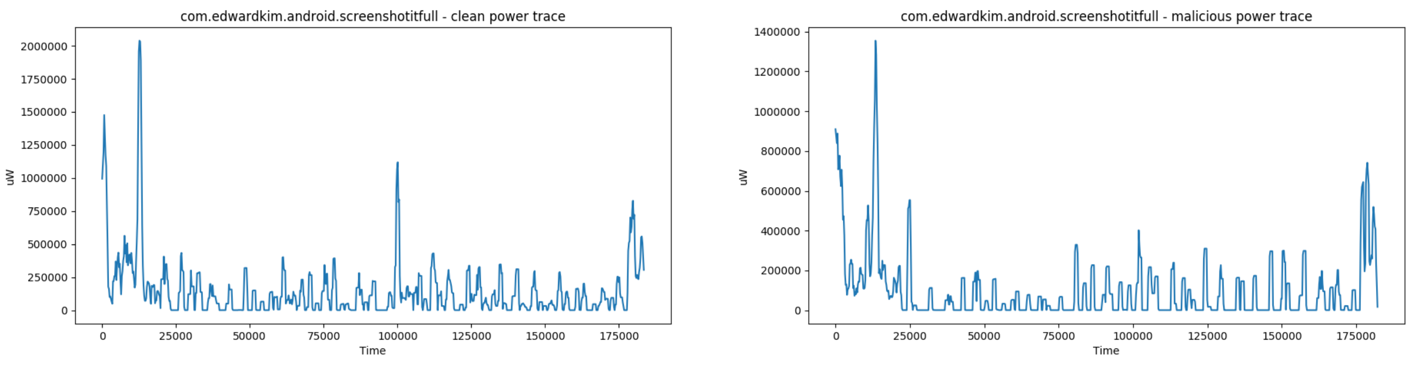
(a) Clean trace



(b) Malicious trace

Figure 3.2: net.lucky.star.mrtm

It should be noted that the fourth app that was tested did not activate, it did not provide a prompt to ask to install the extra app that BaseBridge normally installs. This can also be seen in the power trace, since in the power traces below, in figure 3.3 we can see that both the clean and malicious power trace have a power consumption that is very much alike. It can also be seen that different versions do not always have to result in different power traces. In this case the clean app had a version of 3.35, while the malicious app a version of 1.991.

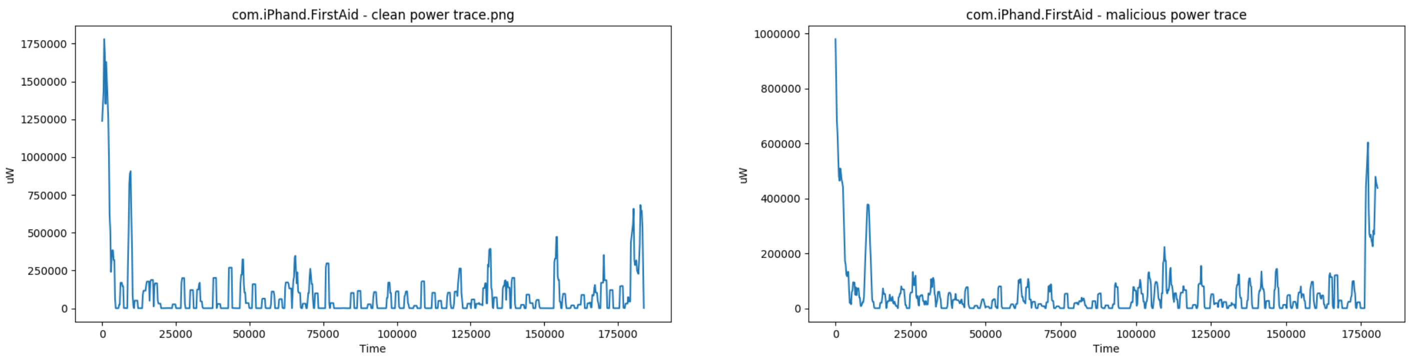


(a) Clean trace (b) Malicious trace

Figure 3.3: com.edwardkim.android.screenshotitfull

Beanbot

Beanbot is another malware that did not show a great increase in power usage. This can be explained because of the initialisation process of the malware, which does not entail any obfuscation or downloading of .apk files. This can be seen in the graphs seen below.



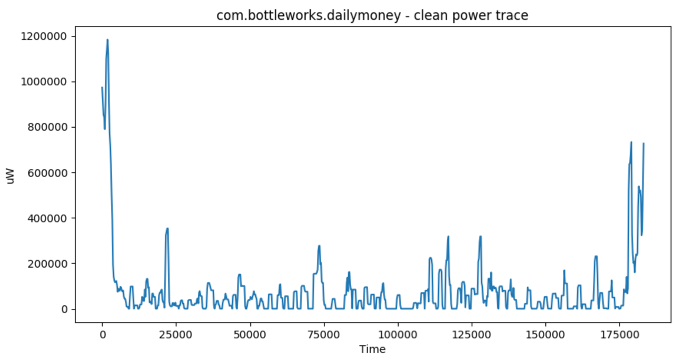
(a) Clean trace (b) Malicious trace

Figure 3.4: com.iPhand.FirstAid

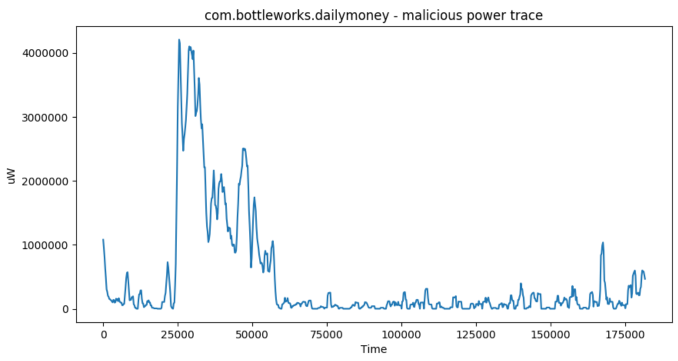
Although we can see an increase in power around the time of the trigger, the same holds for the clean power trace. The clean power trace shows a larger spike than the malicious power trace.

DroidKungFu

This malware shows a clear difference in power usage across the different power traces. Among the different variations of the DroidKungFu malware, all of the clean and malicious trace pairs that have been taken show a clear difference in power consumption. All of the malicious power traces of the DroidKungFu family show a significant increase after the malware has been activated. Below we show one pair of power traces, however, in the appendix A.1.9 the other power traces are listed.



(a) Clean trace



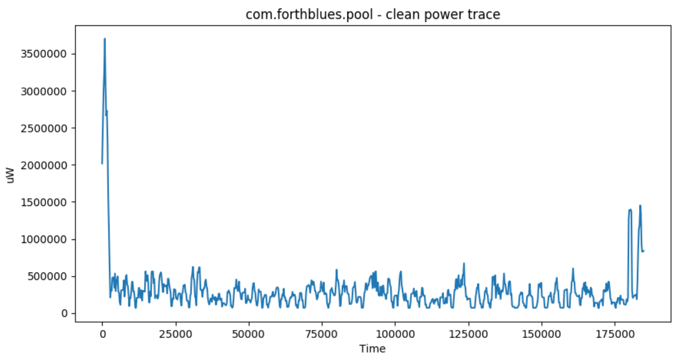
(b) Malicious trace

Figure 3.5: com.bottleworks.dailymoney

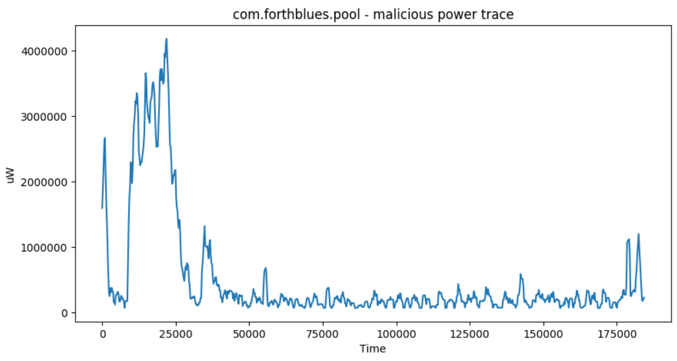
As shown above, we see a significant increase in the power usage. When we look at the shape of the graph, we see that the DroidKungFu variant has a different shape, namely that there is a large increase of power consumption for about 30 seconds. When we compare the average power usage from 25 seconds up to and including 55 seconds, we see a significant difference. The clean trace has an average consumption of 49411 microwatt, while the malicious power traces has an average of 2051842 microwatt. Another average of power consumption in the power traces, from 90 seconds up to and including 120 seconds, shows a smaller difference. The clean power trace's average is then 51016 microwatt, compared to 70000 microwatt for the malicious trace. The percentage difference for the averages from the first interval is 4053%, compared to 37% for the second interval.

Fakeupdates

For FakeUpdates there was one sample. However, there is a clear difference in the power traces. If we look at the clear power trace of figure 3.6, we see a relatively consistent power trace. The power usage does not exceed 500.000 microwatt. When we compare this with the malicious power trace of figure 3.6, we see that around the 10 second mark, there is an increase in power usage.



(a) Clean trace



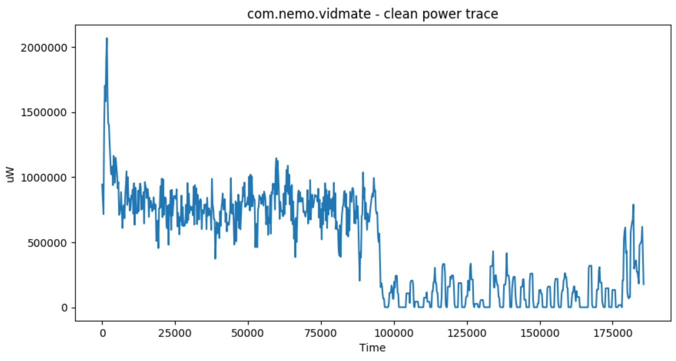
(b) Malicious trace

Figure 3.6: com.forthblues.pool

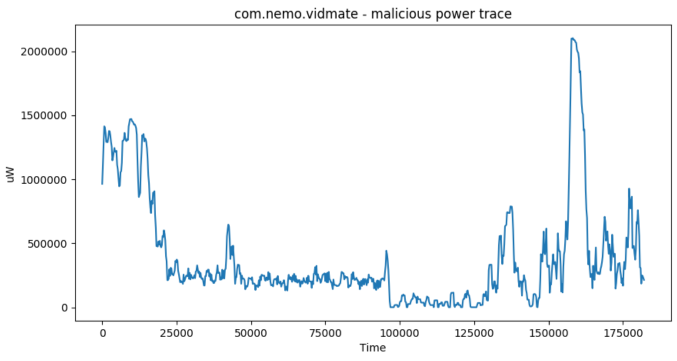
Although the versions do not match, the average usage from 10 seconds up to and including 30 seconds for the clean trace is 291386 microwatt, while this is 2456644 microwatt for the malicious power trace. To compare this difference with other averages in the power trace: the average power usage from 90 seconds up to and including 120 seconds is 236140 microwatt, compared to 175383 microwatt for the malicious power trace. This is a difference of an increase of 743% versus a decrease of 26%, respectively.

Gorpo

Gorpo is malware that triggers when the app is started. The differences in power consumption can be seen in the figure below. It should be noted that besides the peaks in the power trace, the power usages do not match.



(a) Clean trace



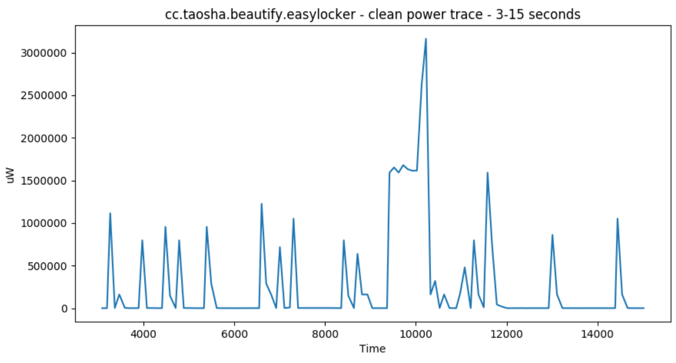
(b) Malicious trace

Figure 3.7: com.nemo.vidmate

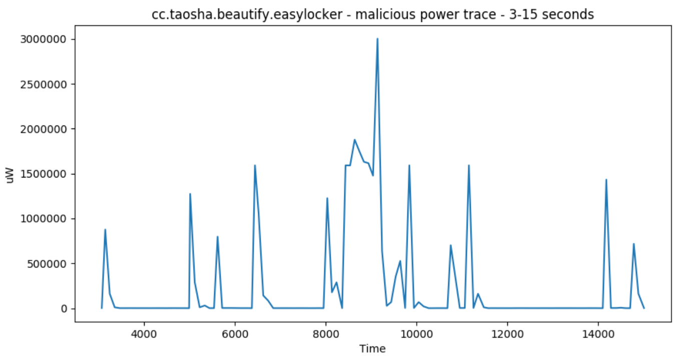
In figure 3.7a we can see firstly the app is using a relatively high amount of energy, which stops around 95 seconds. This same behavior can be seen in figure 3.7b, however, the constant energy usage that is seen in the clean power trace is almost twice as high, compared to the malicious power trace. This difference shows what different app versions can cause. What can be done, however, is look at the shape of power traces. The malicious power traces of Gorpo show a pattern of higher power usage at the beginning of the launch of the app, with another increase in power usage between the 60 and 150 seconds.

Kemoge

Kemoge will unpack and decrypt a .apk file when it is triggered. This means we would expect to see an increase in power consumption around the 10 second mark. However, when we look at the power trace pair below, we see in both traces a peak. The decryption process in the first sample takes one second. Broadcasting the intent when the clean app is running shows the same amount of energy consumption. Below in figure 3.8 we can see one small difference between the two peaks. The malicious power trace contains a peak within a peak.



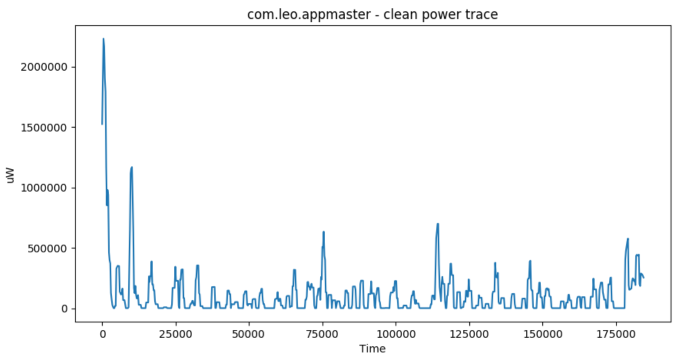
(a) Clean trace



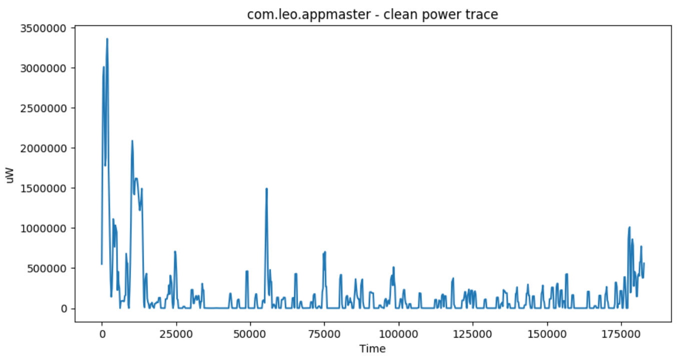
(b) Malicious trace

Figure 3.8: cc.taosha.beautify.easylocker

The second malicious sample took longer to decrypt the .apk file. This is also seen in figure 3.9 below. The malicious power trace shows a wider power peak, compared to the clean power trace.



(a) Clean trace

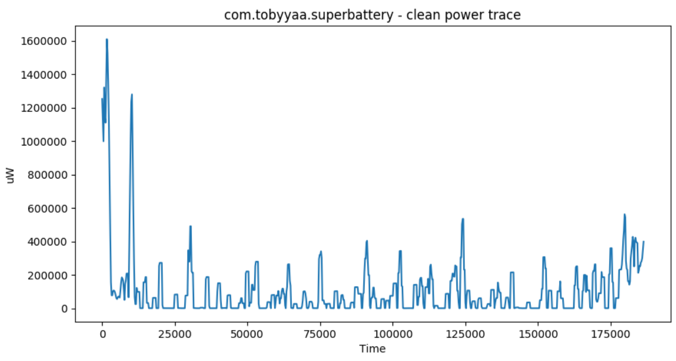


(b) Malicious trace

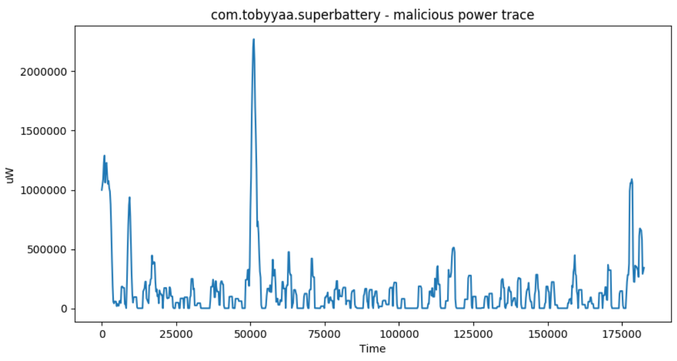
Figure 3.9: com.leo.appmaster

Pjapps

Pjapps is a malware family that is not packed with another .apk file, it also does not use heavy obfuscation. This may explain why the power traces of Pjapps do not show any significant increase in energy consumption. Below are the two pairs of power traces.

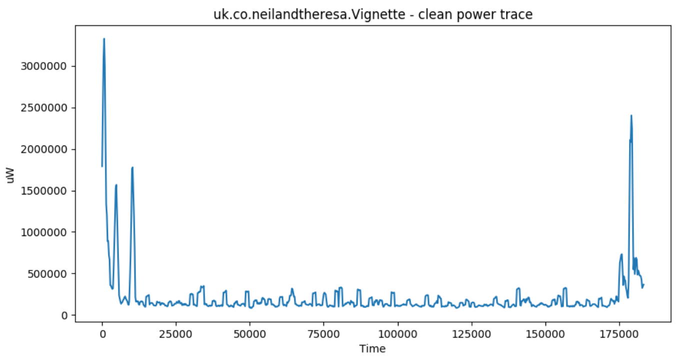


(a) Clean trace

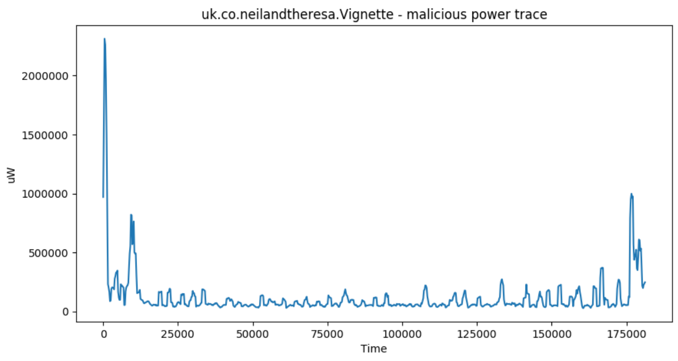


(b) Malicious trace

Figure 3.10: com.tobyyaa.superbattery



(a) Clean trace



(b) Malicious trace

Figure 3.11: uk.co.neilandtheresa.Vignette

In the first pair there seems to be an increase in energy consumption. However, when we look at the second pair of power traces, the increase is less significant. There are also no other increases that seem consistent among the two malicious power traces. Due to the difference in versions of the app, we cannot say with confidence that Pjapps shows clear differences in the power traces.

Chapter 4

Related Work

This chapter will discuss some of the work that is related to detecting malware via energy consumption traces. A research paper from 2016 did a literature study on the different research on anomalies in power consumption for malware detection [22].

The paper described different conclusions about anomaly detection based on power consumption. For instance, it talked about the differences in power needs for different kinds of apps. For instance, a news app would require different energy needs compared to a game or media app. Since we are only using repackaged apps in our research, it is possible to create a so called clean power trace, which would match the normal power consumption needs of the app. The then obtained future traces could be matched with the original power trace. The paper also describes the various factors that can influence the detection by power consumption. The state of the battery, such as age, health and temperature influence the accuracy of the power traces. In addition, the start state of the device, on which the power trace is taken, should be replicated as close as possible each time a power trace is taken.

It further concluded three benefits that power consumption-based malware detection could offer. Firstly, it mitigates the metamorphism of malware. Metamorphism changes the signature of the malware, ¡which is meant to defeat signature-based malware detection. Secondly, the energy-based detection uses a whitelist instead of a blacklist. Instead of having a blacklist with signatures of programs that are not allowed to be installed on the system, there is a whitelist with signatures that are allowed. The author argues that this is highly desired for resource constrained devices, since the whitelist is much smaller than the blacklist. Lastly, the author states that using a whitelist allows the detection mechanism to detect unknown malwares, that differ significantly from the whitelist, something that is relevant for repackaged malware.

A research paper about power consumption-based malware detection

[34] created six different power profiles related to apps. These profiles were Games, Internet, Idle, Malware, Music, and Multimedia. It should be noted that the malware profile in this paper was standalone malware. This meant that the profiles of the malware all had something in common, namely that they were sitting idle and would activate by certain events. The difference with repackaged malware, compared to standalone malware, is that there is no common power profile, because that depends on the app the malware is packaged in. However, the malware that is dropped upon execution of repackaged apps could share a general power profile.

Hoffmann, Neumann, and Holz researched malware detection with their own variant of a malware [32].The dummy malware was able to send and receive SMS, communicate via web sockets, and access contacts and other data on the phone that is often accessed by malware. They concluded that software-based approaches for measuring power consumption on Android devices is not satisfactory in most cases. They state that a system that uses power consumption based data retrieved from software power monitors is infeasible. They write that such a system would generate a relatively high amount of false-positives and that creating a precise power profile for an app is very complex. The authors suggest that a malware that will cause a power consumption increase of below 2% will often go undetected, due to error rates of software power monitors.

There has also been research done on crypto-ransomware on IoT devices [26]. The researchers used power consumption information to detect ransomware. The researchers used PowerTutor and devices from which the power traces were taken had Android 4.4 installed on it. Another paper did research on power consumption in covert channels [25]. The researchers created a framework that used neural networks and classification trees to determine whether two applications were communicating covertly, e.g. via different intents or file sizes.

In summary, different conclusions have been drawn on the use of the power consumption for malware detection. However, there has been a lot of research on power traces to detect malware. It also seems to be rather complex to create power signatures that will provide a way to successfully detect malware. In this thesis we will provide one extra characteristic of malware that could be used to detect malware more easily, namely the initialisation phase. So far, we have not read any literature that specifically writes about the initialisation phase as being an interesting phase to take a power trace of.

Chapter 5

Conclusions

5.1 Discussion

We have observed that in some cases, there were significant differences in the malware's power trace compared to the clean power trace. This is partly because of how the malware works, but it can also be because of the app's power behavior. Malware that is repackaged with apps that require a relatively low amount of energy are more easy to spot. The contrary holds for energy needy applications.

Some of the apps used in this thesis are relatively old, which could misrepresent the power traces for more modern apps. However, the current trend of malware [20] still uses exploits, or tries to install malicious .apk files, which was also the case with the malware used in this data set.

In this thesis software power monitoring has been used, together with apps that mostly did not match versions. Because of this, it is harder to conclude on the average increase the malware could have on the app's power consumption. This is due to different versions of apps having different power consumptions, and because software power monitoring is less reliable compared to hardware power monitoring [29].

Although the malware that was activated in the experiment required prior knowledge of the malware, it does show that upon activation there are significant changes in power consumption, compared to the clean power trace. In this case, the power trace could help in confirming that something is wrong with a .apk file. This is for the scenario in which an antivirus solution has a hunch about a certain malware. In that case it can try to activate it and compare the power trace with the clean app. This would require that the actions to activate the malicious .apk file should be replicated for the clean app. It should be noted that for this situation it is assumed that there is a cloud solution, where users upload .apk files to the cloud, or provide a URL from which the server can download the .apk file.

Additionally, there was malware, such as Gorpo and FakeUpdates, that required only running the app once in order to activate. Other likewise malware could also show up in the power trace, if the change is significant enough.

5.2 Conclusion

In this thesis we have shown that the phase in which the malware infects a device can be used in malware detection by energy consumption. The moment malware infects a device, there is often communication with a command-and-control server, it tries to launch certain exploits, or retrieves .apk files. These actions all have an effect on an app's power consumption. We have observed that in some cases these initialisation phases show significant increases in the power trace, compared to the same app without the repackaged malware. However, some malware did not show significant differences in the power traces that were retrieved during the experiment. The results of this thesis can help in creating a more extended power profile for an app. Besides looking at the average increase of an app's power usage, we can also look at the power usage when the app is first launched, or when a certain intent is sent to the app.

Bibliography

[1] Androl4b. https://github.com/sh4hin/Androl4b.

[2] dex2jar. https://github.com/pxb1988/dex2jar.

[3] Frida. https://www.frida.re/.

[4] Java Decompiler. http://jd.benow.ca/.

[5] Powertutor.

[6] Trepn Power Profiler - Qualcomm Developer Network.

[7] VirusTotal. https://www.virustotal.com, accessed on 10 July, 2018.

[8] Which mobile devices report accurate system power consumption? - qualcomm developer network.

[9] Wireshark.

[10] Google Code Projects Host Android Malware — McAfee Blogs, 2011. https://securingtomorrow.mcafee.com/mcafee-labs/google-code-projects-host-android-malware/, accessed on 25 June, 2018.

[11] New DroidKungFu Variants – Again!, 2011. https://www.csc2.ncsu.edu/faculty/xjiang4/DroidKungFu3/, accessed on 30 July, 2018.

[12] BeanBot, March 2012. https://www.csc2.ncsu.edu/faculty/xjiang4/BeanBot/, accessed on 25 June, 2018.

[13] Kemoge: Another Mobile Malicious Adware Infecting Over 20 Countries, 2015. https://www.fireeye.com/blog/threat-research/2015/10/kemoge_another_mobi.html, accessed on 5 July, 2018.

[14] Taking root - Securelist, 2015. https://securelist.com/taking-root/71981/, accessed on 5 July, 2018.

[15] How to protect vs. wannacrypt - Kaspersky Lab official blogog, May 2017. https://www.kaspersky.com/blog/wannacry-ransomware/16518/, accessed on 14 August, 2018.

[16] New DroidKungFu Variants – Again!, 2017. `https://www.zscaler.com/blogs/research/malicious-android-ads-leading-drive-downloads`, accessed on 14 August, 2018.

[17] New Petya / NotPetya / ExPetr ransomware outbreak - Kaspersky Lab official blog, 2017. `https://www.kaspersky.com/blog/new-ransomware-epidemics/17314/`, accessed on 14 August, 2018.

[18] GhostTeam Adware can Steal Facebook Credentials - TrendLabs Security Intelligence Blog, 2018. `https://blog.trendmicro.com/trendlabs-security-intelligence/ghostteam-adware-can-steal-facebook-credentials/`, accessed on 30 July, 2018.

[19] Hidden App Malware Found on Google Play — Symantec Blogs, May 2018. `https://www.symantec.com/blogs/threat-intelligence/hidden-app-malware-google-play`, accessed on 30 July, 2018.

[20] Mobile malware evolution 2017 - Securelist, 2018. `https://securelist.com/mobile-malware-review-2017/84139/`, accessed on 14 August, 2018.

[21] Mobile Operating System Market Share Worldwide — StatCounter Global Stats, 2018. `http://gs.statcounter.com/os-market-share/mobile/worldwide`, accessed on 14 August, 2018.

[22] Jameel A. Qadri, Thomas M. Chen, and Jorge Blasco. A Review of Significance of Energy-Consumption Anomaly in Malware Detection in Mobile Devices. October 2016.

[23] Albano, Pietro and Castiglione Aniello and Cattaneo Giuseppe and Santis De, Alfredo. A novel anti-forensics technique for the android os. October 2011.

[24] Mohammad Ashraful Hoque, Matti Siekkinen, Kashif Nizam Khan, Yu Xiao, and Tarkoma Sasu. Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices. *ACM Computing Surveys*, 2015.

[25] Amin Azmoodeh, Ali Dehghantanha, Mauro Conti, and Kim-Kwang Raymond Choo. Seeing the Unseen: Revealing Mobile Malware Hidden Communications via Energy Consumption and Artificial Intelligence. *IEEE Transactions on Information Forensics and Security*, 11:.799–81, April 2016.

[26] Amin Azmoodeh, Ali Dehghantanha, Mauro Conti, and Kim-Kwang Raymond Choo. Detecting crypto-ransomware in IoT networks based on energy consumption footprint. *Journal of Ambient Intelligence and Humanized Computing*, 9:1141–1152, August 2017.

[27] Byrd, Brittany and Zhou, Bing and Liu, Qingzhong. Android system partition totraffic data? 3(2):37–42, December 2017.

[28] Eric Chien. Motivations of recent android malware. Technical report, Symantec Corporation, Mountain View, CA 94043 USA, 2011. `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/motivations_of_recent_android_malware.pdf`, accessed on 5 July, 2018.

[29] Robertas Damaševičius, Vytautas Štuikys, and Jevgenijus Toldinas. Methods for Measurement of Energy Consumption in Mobile Devices. *Metrology and Measurement Systems*, XX:419–430, September 2013.

[30] Ken Dunham, Shane Hartman, Jose Andre Morales, Manu Quintans, and Tim Strazzere. *Android malware and analysis*. CRC Press, 2015.

[31] Wei Fengguo, Li Yuping, Roy Sankardas, Ou Xinming, and Zhou Wu. Deep Ground Truth Analysis of Current Android Malware, 2017.

[32] Johannes Hoffmann, Stephan Neumann, and Thorsten Holz. Mobile Malware Detection Based on Energy Fingerprints — A Dead End? pages 1–22, October 2013.

[33] Nancy et al. A survey on android malwares and their detection mechanisms. *International Journal of Computer Science and Information Technologies*, 7(4):1779–1782, 2016.

[34] Thomas Zefferer, Peter Teufl, David Derler, Klaus Potzmader, Alexander Oprisnik, Hubert Gasparitz, and Andrea Hoeller. Power Consumption-based Application Classification and Malware Detection on Android Using Machine-Learning Techniques. *Future Computing*, 5:26–31, 2013.

[35] Yajin Zhou and Xuxian Jiang. An analysis of the anserverbot trojan. Technical report, North Carolina State University, 2011. `https://www.csc2.ncsu.edu/faculty/xjiang4/pubs/AnserverBot_Analysis.pdf`, accessed on 26 June, 2018.

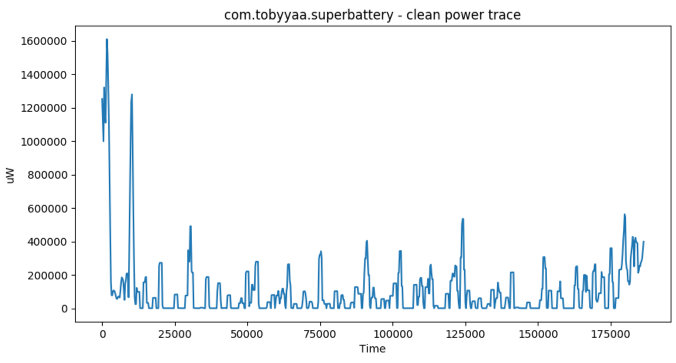
[36] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. *2012 IEEE Symposium on Security and Privacy*, July 2012.

Appendix A

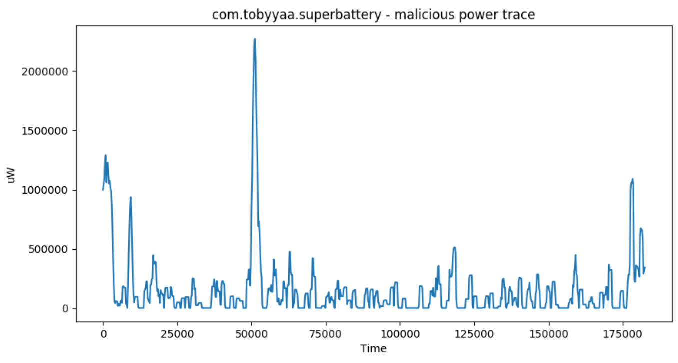
Appendix

A.1 Power samples

A.1.1 Pjapps

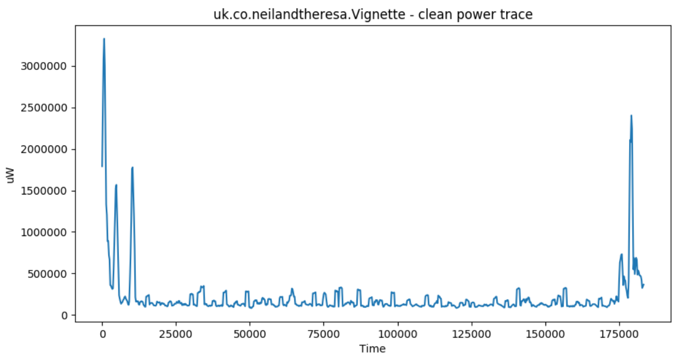


(a) Clean trace

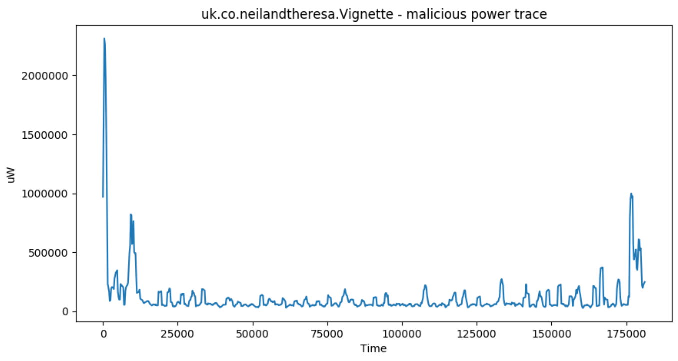


(b) Malicious trace

Figure A.1: com.tobyyaa.superbattery



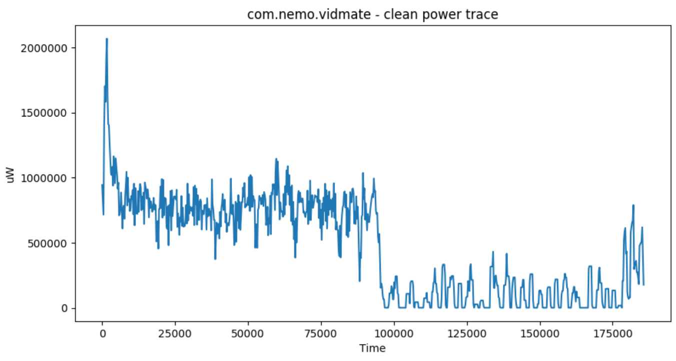
(a) Clean trace



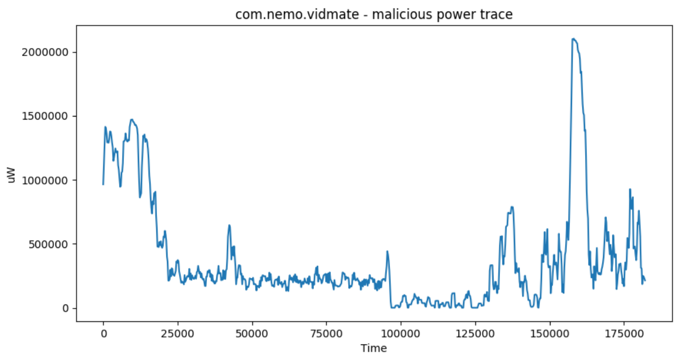
(b) Malicious trace

Figure A.2: uk.co.neilandtheresa.Vignette

A.1.2 Gorpo

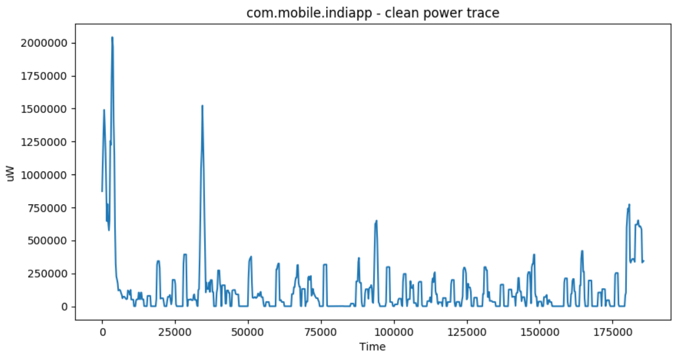


(a) Clean trace

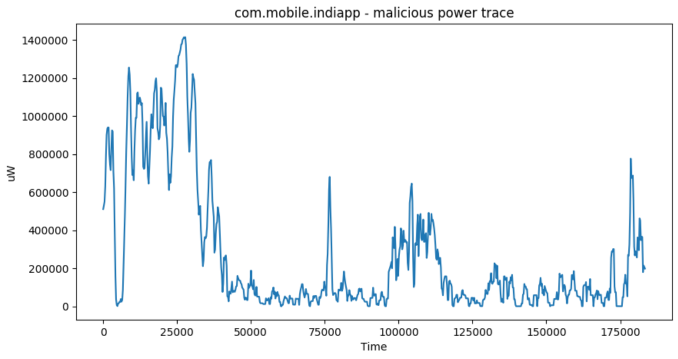


(b) Malicious trace

Figure A.3: com.nemo.vidmate

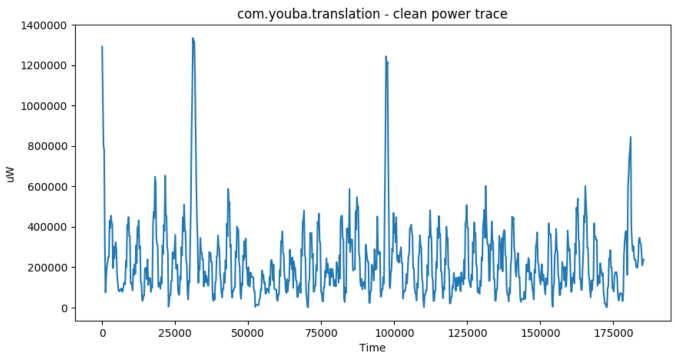


(a) Clean trace

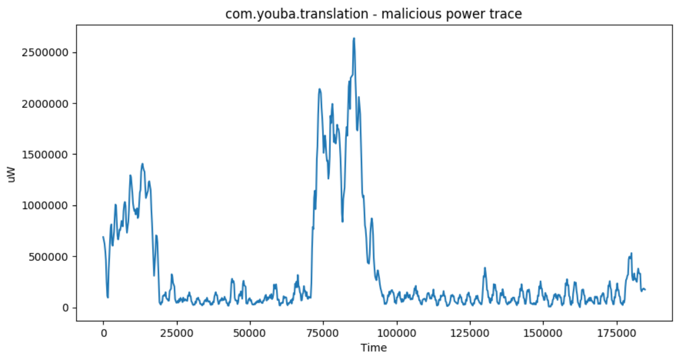


(b) Malicious trace

Figure A.4: com.mobile.indiapp



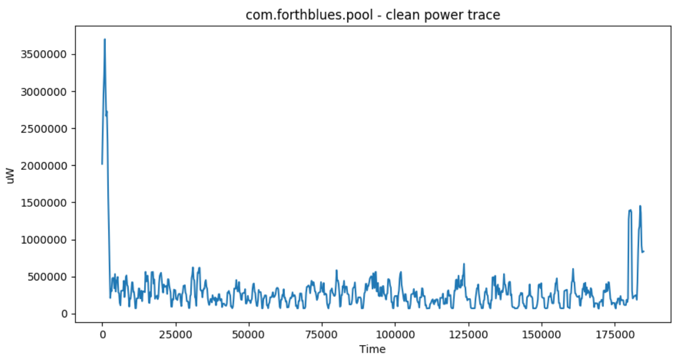
(a) Clean trace



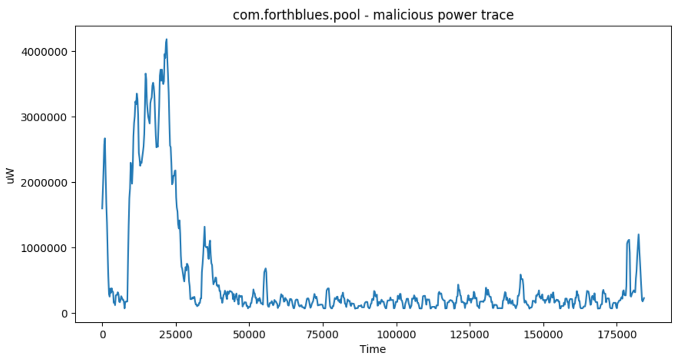
(b) Malicious trace

Figure A.5: com.youba.translation

A.1.3 FakeUpdates



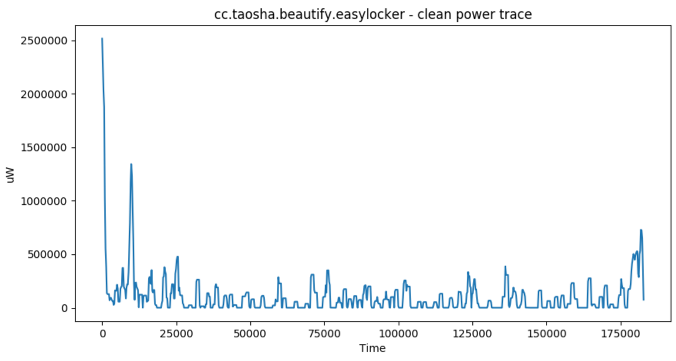
(a) Clean trace



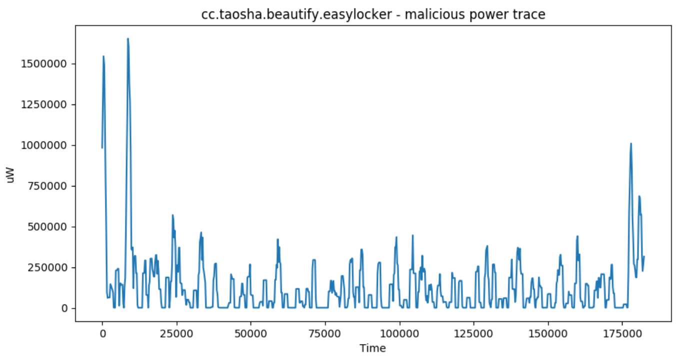
(b) Malicious trace

Figure A.6: com.forthblues.pool

A.1.4 Kemoge

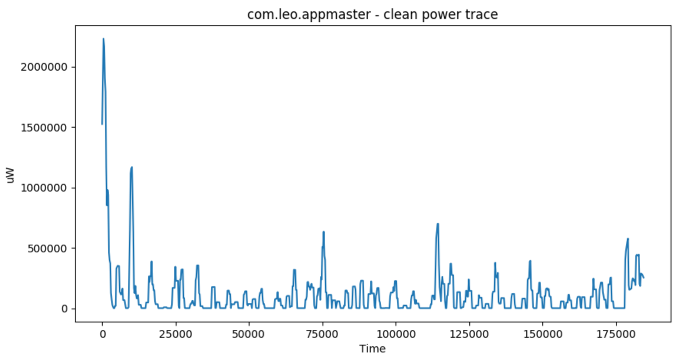


(a) Clean trace

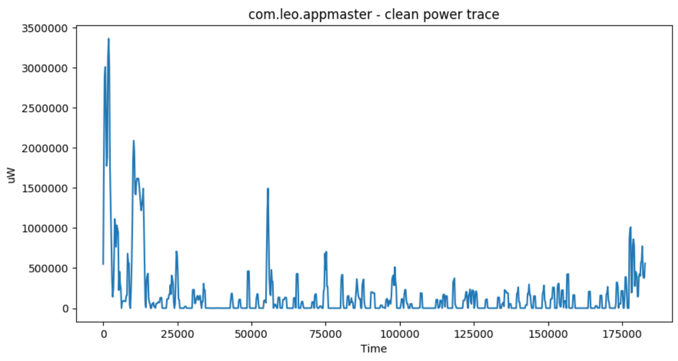


(b) Malicious trace

Figure A.7: cc.toasha.beautify.easylocker



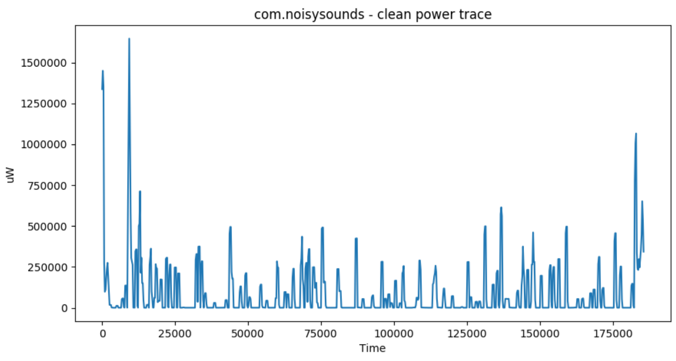
(a) Clean trace



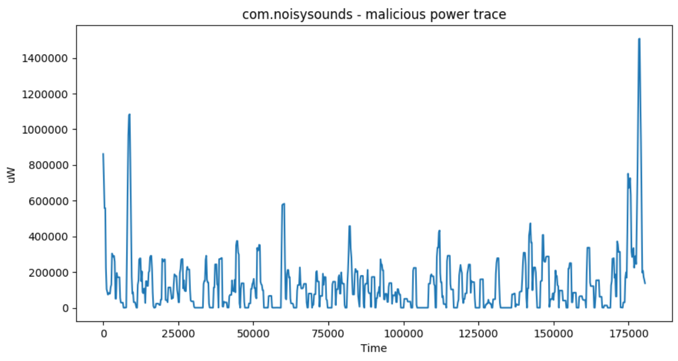
(b) Malicious trace

Figure A.8: com.leo.appmaster

A.1.5 ADRD



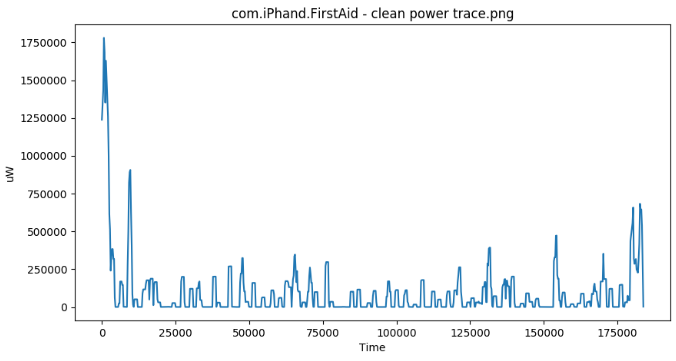
(a) Clean trace



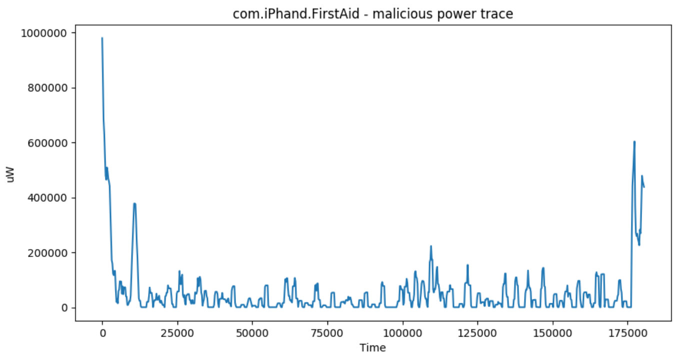
(b) Malicious trace

Figure A.9: com.noisysounds

A.1.6 BeanBot



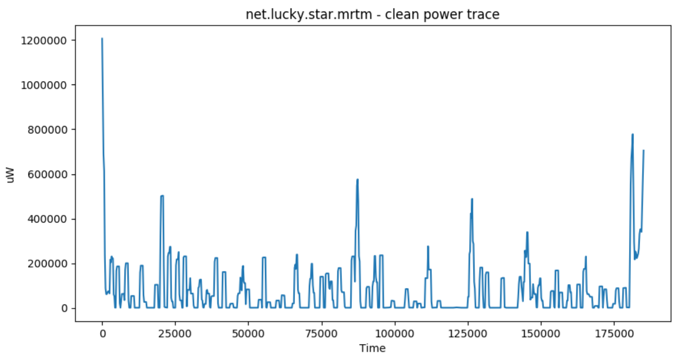
(a) Clean trace



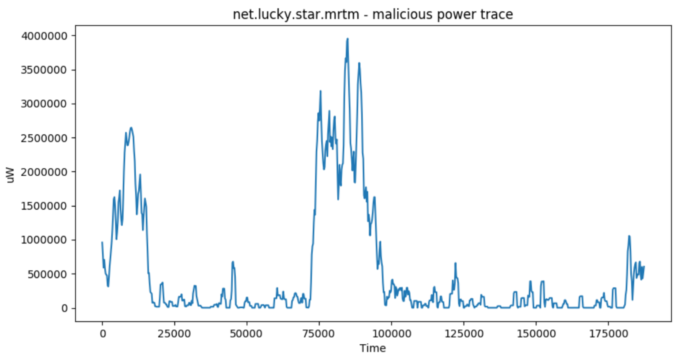
(b) Malicious trace

Figure A.10: com.iPhand.FirstAid

A.1.7 BaseBridge

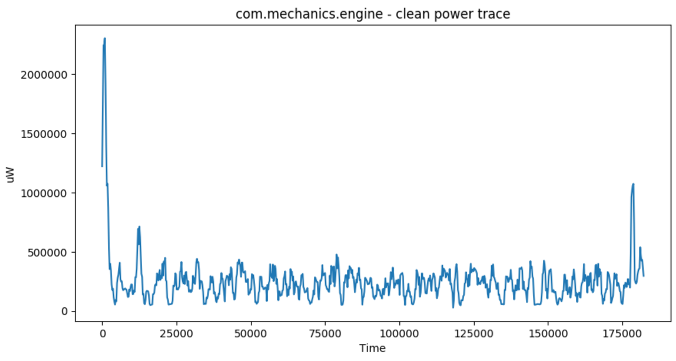


(a) Clean trace

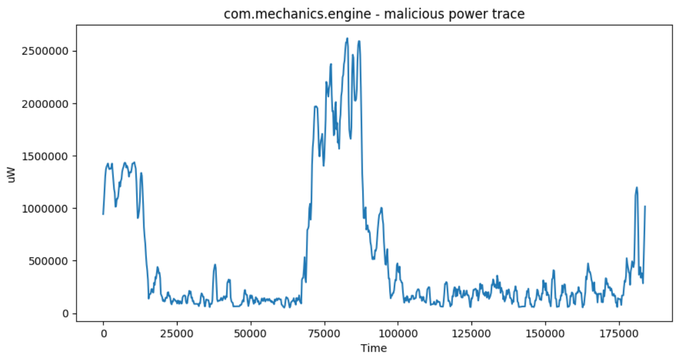


(b) Malicious trace

Figure A.11: net.lucky.star.mrtm

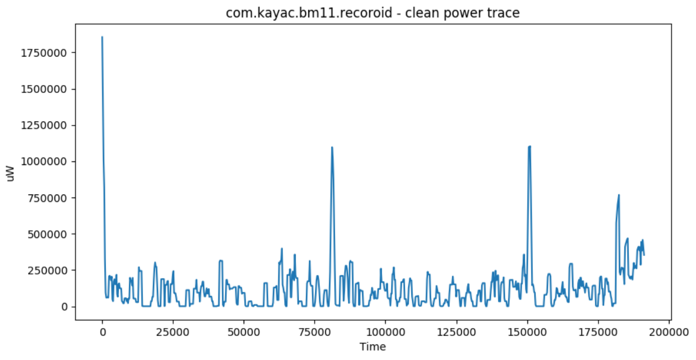


(a) Clean trace

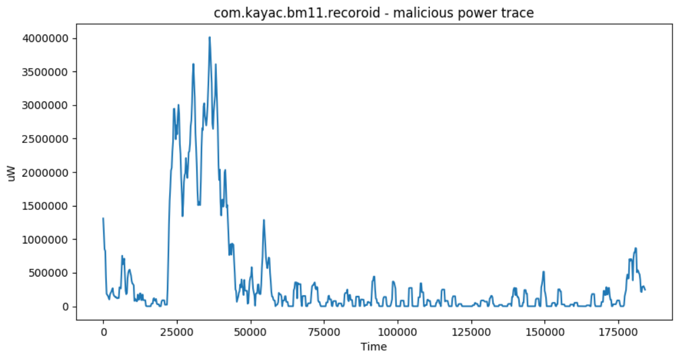


(b) Malicious trace

Figure A.12: com.mechanics.engine

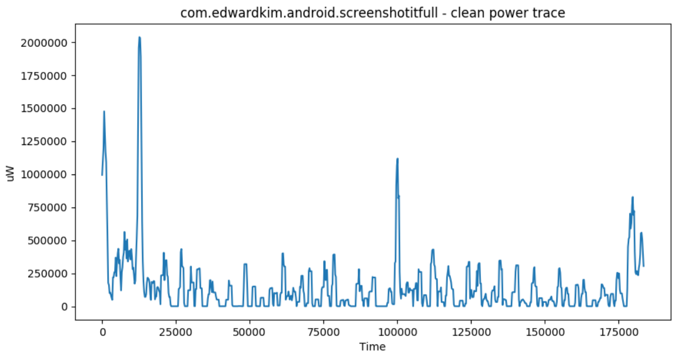


(a) Clean trace

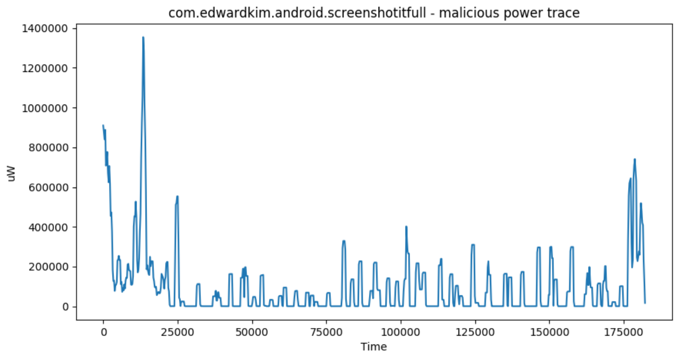


(b) Malicious trace

Figure A.13: com.kayac.bm11.recoroid



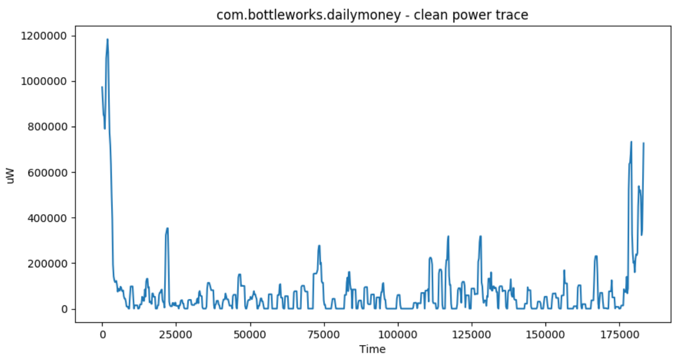
(a) Clean trace



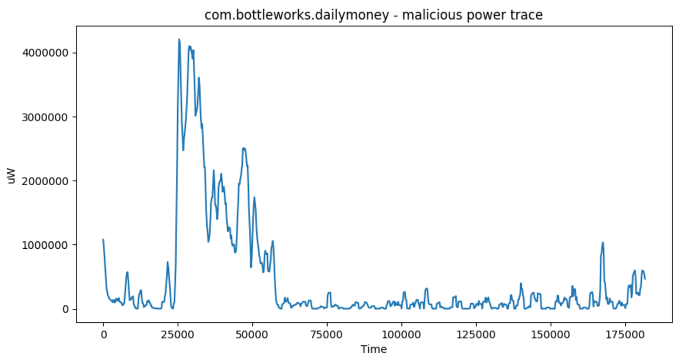
(b) Malicious trace

Figure A.14: com.edwardkim.android.screenshotitfull

A.1.8 DroidKungFu2



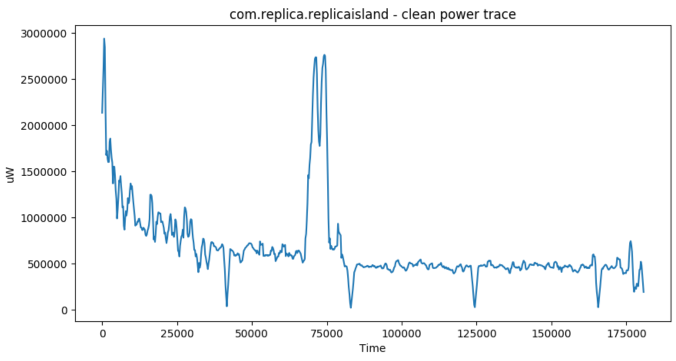
(a) Clean trace



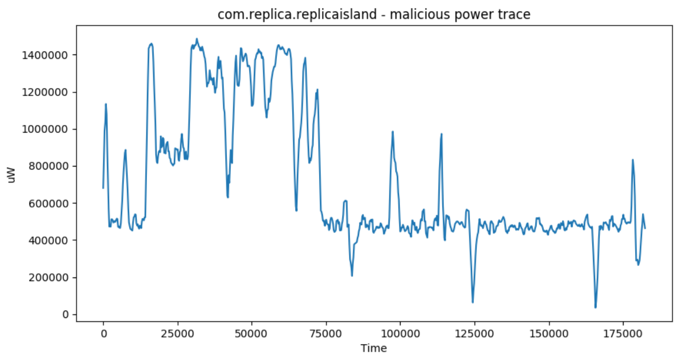
(b) Malicious trace

Figure A.15: com.bottleworks.dailymoney

A.1.9 DroidKungFu3

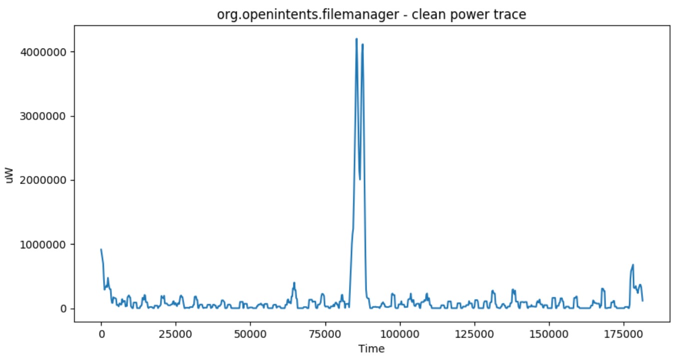


(a) Clean trace

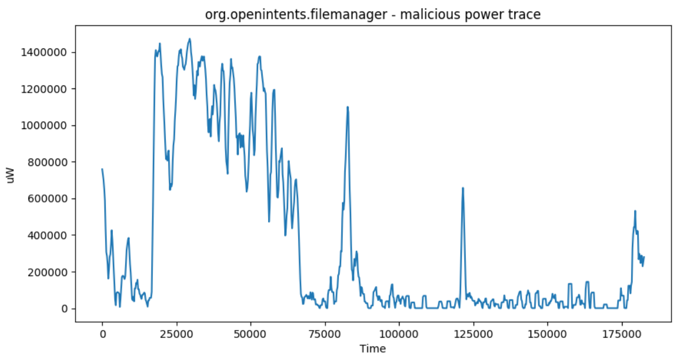


(b) Malicious trace

Figure A.16: com.replica.replicaisland



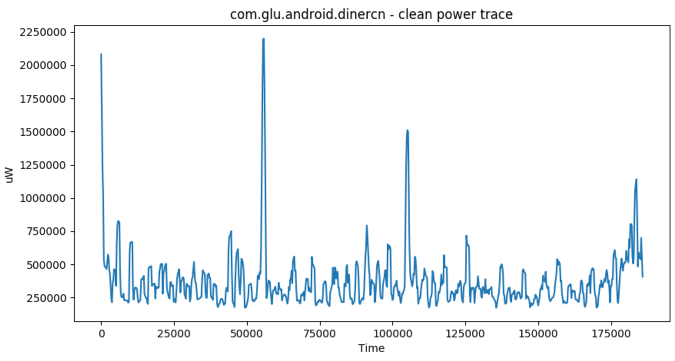
(a) Clean trace



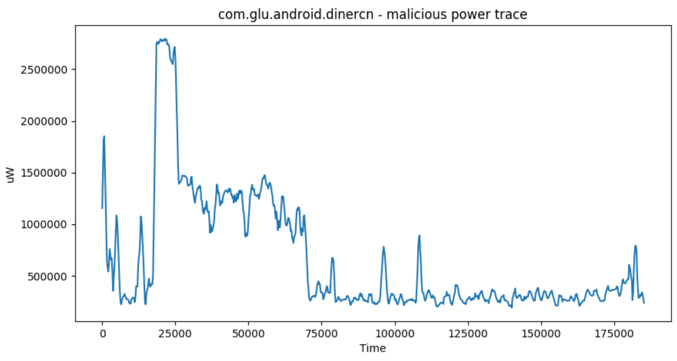
(b) Malicious trace

Figure A.17: org.openintents.filemanager

A.1.10 DroidKungFu4



(a) Clean trace



(b) Malicious trace

Figure A.18: com.glu.android.dinercn