UC Davis
UC Davis Electronic Theses and Dissertations

Title

Utilizing HPCs as a Method for Update Malware Detection

Permalink

https://escholarship.org/uc/item/7012c4cp

Author

Gong, Mina

Publication Date

2021

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Utilizing HPCs as a Method for Update Malware Detection

By

Mina Gong

Thesis

Submitted in partial satisfaction of the requirements for the degree of

Master of Science

in

Electrical and Computer Engineering

in the

Office of Graduate Studies

of the

University of California

Davis

Approved:

Houman Homayoun, Chair

Hussain Al-Asaad

Rajeevan Amirtharajah

Committee in Charge

2021

Copyright © 2021 by

Mina Gong

All rights reserved.

Contents

Abstract iv

Acknowledgments v

1 Introduction **1**

1.1 Application Updates and Associated Malware 1

1.2 Why Hardware Performance Counters? 3

1.3 Contributions 3

1.4 Content Overview 4

2 Background **5**

2.1 Malware Types and Update Malware 5

2.1.1 Malware Types 5

2.1.2 Update Malware 6

2.2 Repackaged and Piggybacked Applications 7

2.3 Hardware Performance Counters (HPCs) 8

2.4 Machine Learning for Classification 9

2.5 Malware Visualization Technique 10

2.6 Related Work 11

2.6.1 Network pattern-based detection 12

2.6.2 Feedback-loop depicting active learning (Aion) 13

3 Experiment **15**

3.1 Testing Environment Setup 15

3.2 Simpleperf 16

3.3 Dataset 17

3.4 Automation of Experiment Pipeline 18

3.4.1 Download APK files from AndroZoo 18

3.4.2 Collect Data with HPCs 19

3.4.3 Transformation of the raw data in understandable format 21

3.5 Machine Learning 22

4 Discussion **24**

4.1 Piggybacked applications 24

4.1.1 Piggybacked Malware: Adwo and Kuguo (Adware type) 25

4.1.2 Piggybacked Update Malware: DroidKungFu and Plankton (Trojan type) 26

4.1.3 Discovery 31

4.2 Evaluation 32

5 Conclusion **38**

Abstract

Utilizing HPCs as a Method for Update Malware Detection

The daily use of mobile phones, and particularly smartphones, has become an integral part of modern civilization. With the continued adoption of smartphones by users world wide, an abundance of applications to meet their various demands is a necessity. A plethora of applications are provided through markets, such as the Google Play Store, that allow users to download applications directly to their device. As the Google Play Store is one of the most popular markets, they provide considerably robust security, and users have trust in their ability to properly vet hosted products. Be that as it may, there exists a subset of society which seeks to exploit and take advantage of unsuspecting victims. Due to the robustness of the security scanning, malware developers must circumvent marketplace security controls. An example of an exploit is called piggybacking. In this case, a benign application can be prepared for an update attack with the piggybacking technique that injects the malicious code. Detecting this change in the application is the main focus of the study. Because of the piggybacking technique, which is cleverly obfuscated, static analysis is not a consistent method to detect malice; hardware performance counters (HPCs) that are capable of dynamic analysis are adopted to explore whether the HPCs have the potential to detect clandestine applications. HPCs were utilized to observe the possibility of detection of piggybacked applications, furthermore, the piggybacked applications that contain the update attacks. HPC data was collected via a rooted phone with automated pipelines, and for the depth of the study, the comparison between static data and dynamic data was provided with visualization, call graphs and scatter graphs. Additionally, a machine learning tool, WEKA, was utilized to discover whether the data can classify the applications into benign or malicious. Six different classifiers are selected, and as a result, the Decision Tree classifiers achieved around 94% to 99% detect accuracy proving that HPCs are a viable method to detect update malware. The result led us to determine whether HPCs are utilizable to detect embedded malware.

Acknowledgments

I would like to express my sincere appreciation to my committee chair, Professor Houman Homayoun, who has provided invaluable guidance and support through the course of study. I would not have successfully completed the study without his help.

I would also like to thank committee members Professor Hussain Al-Asaad and Professor Rajeevan Amirtharajah, who were willing to engage in my research providing wonderful advice for me to strengthen my thesis.

As a collaborator, Professor Houman Homayoun introduced Suraj Kesavan, who is in visualization related field. He was not reluctant to offer me feedback of the framework we both developed and to share resources to enable further discovery. He also did a fantastic job of integrating the visualization part, as well as making the framework smooth by adding the final touches.

Lastly, I would like to thank our family and friends who always give infinite support and emotional relief during the tough time.

Chapter 1
Introduction

1.1 Application Updates and Associated Malware

The smartphone market has become saturated with a plethora of options; manufacturers seeking to remain competitive keep producing updated designs nearly every year. As with smartphones, the number of applications available for download in application markets is continuously growing to meet the demand for new and updated products. In order to provide the service continuously no matter what new features new smartphones have, applications should be designed to stay current with new functionalities or specifications. Fortunately, developers use an application update mechanism to support newly added devices via Android marketplaces. At face value, the update mechanism seems to solve the requirement of the application remaining current. The unfortunate reality is that malevolent application developers seek to steal confidential and sensitive user data through the same mechanism.

When users download applications directly from a trusted source, such as the Google Play Store or Samsung's Galaxy Store, they are able to obtain many applications and can reasonably assume the applications are free of malware. However, there is a possibility a user may want to download an application (.apk file) from an untrusted source. To do so, users need to enable "trust unknown sources" or something similar depending on the smartphone, typically found in the security tab of the smartphone's settings. This option was used in this research to download untrusted applications for research purposes.

In this case, malicious applications could be downloaded immediately without any sort of security check. In contrast, well-known and frequently used Android marketplaces provide a respectable degree of security examination before the application is hosted for download. Therefore, applications that have been vetted by the marketplace and have been installed on a user's smartphone are considered to be safe, and these applications obtain trust between the user and where the application is hosted. Attackers may exploit this trust by providing a malicious application update.

If an update occurs via Android marketplaces, as mentioned above, the marketplaces regularly scan applications to search for the presence of malware. Though the scanning does reduce the number of affected applications, there exists methods that can assist in evading detection by the security scanner. Examples include utilizing time delay to avoid security checking when registering an updated malicious version of a benign application, as well as accumulating the permissions of every benign version update for hostile usage later [1]. Similar to varying methods of evasion, application update attacks can occur in different ways. One way is that nefarious developers inject virulent segments into an existing application. Another way is that they modify the code and re-register it to a market. Lastly, when a user starts an application, the application is connected to the server, finds the differences between current file and updated files, and updates partially while the user uses the application with full connection of the Internet. The first case refers to piggybacked applications, the second case refers to repackaged applications [2], and the latter refers to malware utilizing the update mechanism. Anserverbot and Plankton are well known examples of this type of malware, where some of the update attacks display a bogus update window to allure users to accept [3, 4]. The piggyback technique could be considered to be a subset of the repackaging technique, and our focus is on piggybacked applications to test the possibility of using HPCs to detect the update malware. Distinguishing whether an application exhibits benign or malicious behavior via piggybacking presents a challenge for the following reason: malicious source code is injected and hidden within the benign package, making detection difficult using traditional scanning methods. Malevolent developers use conditional variables stacked to wrap the

malicious part safely, often evading detection by not allowing the malicious route to be triggered instantaneously. Therefore, the application might not act differently from the benign version and the scanner might not detect the presence of malware. This is one of the reasons why malware can remain hidden for a long time and detection is unlikely when the target notices no abnormal behavior. To prevent this, detecting malware as fast as possible after introduction to a system and supporting treatment based on analysis other than the given security scanning are equally important.

1.2 Why Hardware Performance Counters?

Some of the malware that is embedded within a legitimate application is obfuscated and of an inconspicuous design; this is why some of the update malware is challenging to detect with static analysis. Because of this, a different method was required, which is satisfied by using dynamic analysis. Many tools and methods exist for dynamic analysis; Hardware Performance Counters (HPCs) were chosen since the author has the prior research utilizing HPCs with the x86 Intel IvyBridge Processor. Moreover, after observing previous work with successful results in detecting malware with HPCs on the x86 architecture [5], pursuit of this topic of research was inspiring. Mostly, update attacks require the internet connection for the malevolent purpose, which cannot be one of the requirements for the upcoming experiment due to both known and unknown risk. Even though the network service cannot be used, the attempt to connect to the server suspiciously and more frequently than the normal version of applications can be caught within the hardware events, such as an extreme increase in usage of cache, and occurrence of cache misses. Therefore, HPCs are utilized to detect the update malware, more specifically, piggybacked update malware. The goal of the research is to determine the feasibility of using these counters on them.

1.3 Contributions

Thus, our approach is to use HPCs in ARM architecture phones to observe the usability and possibility for distinguishing the benign application and piggybacked malicious application. In the process of collecting the dataset, the number of piggybacked

applications are limited compared to the regular malware applications. Out of the available malware databases, AndroZoo was chosen since they have an enormous collection of Android applications, including piggyback type malware, and is actively managed compared to other collections. With the dataset, each step, from downloading piggybacked applications specifically to transforming the collected data to a comprehensive format for the machine learning step, is time-consuming and complicated. Therefore, the automated pipeline is desirable. The automated items will be covered in detail in the corresponding section. Furthermore, the outcome of using different Machine Learning classifiers with collected HPC data will be discussed at the end. To the best of my knowledge, applying HPCs for the detection of update malware, specifically piggybacked applications, is being explored for the first time.

1.4 Content Overview

Chapter 2 covers the background knowledge to understand what update malware, piggybacked applications, and Hardware Performance Counters (HPCs) are, what machine learning classifiers are useful in malware detection, and introduces some of the papers that conducted different techniques in malware detection. Chapter 3 covers the environment setup for the experiment, a tool, called Simpleperf, which assists the manipulation of HPCs, dataset, automated pipelines, and Machine Learning classifiers adopted in this paper. Chapter 4 discusses the results based on visualized data and summarizes the evaluation of the classification, and Chapter 5 finalizes the thesis with the conclusion.

Chapter 2
Background

2.1 Malware Types and Update Malware

2.1.1 Malware Types

As the number of applications grows, as does the footprint attackers have for immense opportunities in exploitation. As proof, many different names and types of Android malware exist, such as Smspay and Artemis as Riskware, DroidKungFu and Plankton as Trojan, leaving Dowgin, Adwo, and Kuguo in the Adware category. The following is a brief description of what each category of malware entails. Trojan refers to the malware that contains a malicious hidden payload within a seemingly legitimate application, which may allow an attacker to gain control of the device and all the data within it once the malware is activated. Adware is a type of advertisement that could be invasive and bothersome due to the pop-up style of the ad alerts. The developers of Adware earn money by supplying advertisements to personal devices. Furthermore, the unintentional agreement of downloading off a platform related to the ads might have exposed an unknown risk to allured users. Another category of malware is Riskware, which is not severely harmful for users as compared to other categories. It violates the application environmental policies, such as affecting other applications, providing a route for other malware to invade, or illegal permissions to use in a user's country.

2.1.2 Update Malware

Plankton, AnserverBot, BaseBridge, and DroidKungFu are referred to as the names of the specific update malware type [3]. These update attacks download a typical update segment embedding the malicious payload at runtime [4]. BaseBridge and DroidKungFu are the same type of malware but the difference between them is that BaseBridge holds the updated version whereas DroidKungFu approaches the updated version via network connection, which will ask users for approval. The stealthier versions are Plankton and AnserverBot because they bypass the agreements from users. Plankton uses a remote server to download a jar file whereas Anserverbot fetches commands from a public blog that has the update version [6]. The details of the update attack techniques are presented in [7]: One way is to permit the execution of the code that was not initially included in an application by loading compiled Android code (i.e. executable DEX files) using Android's DexClassLoader class. A .dex file is created by converting the Java Bytecode, which is formed by compiling an application written in JAVA, to Dalvik Bytecode, which forms Dalvik executable file (.dex). Another way is to download a binary shared object file (.so library) or an executable file containing native code, executed at runtime using Java's Runtime class. The other way is to download a .mp3, .jpg, .flash, or .pdf file having a malicious payload at runtime and execute the files by targeting vulnerabilities in the system libraries. It is not only malware tailored specifically for update attacks; regular benign applications can also be modified for use in update attacks. Attackers can reverse engineer a benign application, modify some parts of the existing code in such a malicious and secretive way that it will pass the security examination, and then recompile and upload the repackaged version to a market. With respect to the study of malware, the above describes the origin of the term *repackaged applications*, which will be discussed in depth later on. Users will not notice the difference between the two different versions of the application using their observation alone. For that reason, repackaging and piggybacking techniques can assist benevolent applications to contain the update attacks.

2.2 Repackaged and Piggybacked Applications

The nature of repackaged and piggybacked applications are different from other regular malware where developers purposefully produce applications to be malicious whereas the repackaged and piggybacked ones are slightly transformed from benign to malicious. [8,9] describes the classification of two terms, and *piggybacking* is a subset of *repackaging*. The difference is that the technique being used for repackaged applications in detail is that a reverse engineering tool is used to decompile an .apk file originally running benevolently, apply slight modifications, recompile, and re-sign them to upload to a marketplace [2], while piggybacked applications inject malicious payload segments, a so called *rider* [10].

A malicious payload can be composed of two ways: the explicit way or implicit way. The explicit way entails payloads that are integrated into the logic of the existing code. In other words, the functionality can be disfigured to some extent from the original behaviors. The implicit way is to use the conditional variables to conceal the malicious code execution. In this case, the code might not be triggered for a long time unless the condition matches what the developer intended. For example, the malicious payload might be executed at a specific time or date or at a specific place. The usage of the two different nature of payload is dependent on what the developers consider important. If they desire to take advantage of victims immediately, then they will go for the explicit path. In contrast, if the developers prefer to not get caught and slightly steal the user information, they would choose the implicit path.

The following is an example of using the repackaging technique to infect an application with Trojan and how long the disclosure took: the relatively popular Barcode Scanner application turned out to be infected with Trojan malware [11]. At some point, the publisher changed and the application was updated at least five times after the original publisher, which includes the nature of the repackaging technique: resigning and distribution. The updated fifth version was revealed to contain a malicious payload. Moreover, disclosing the malware took approximately one month, which proves that repackaged applications are difficult to detect and can be present even in a major Android marketplace.

In this research, I concentrated on piggybacked applications, based on the label by AndroZoo, where the technique is a subset of repackaging. Additionally, for the clarification on DroidKungFu and Plankton malware, which have regular and piggybacked update attack versions, the application collection that is used for the experiment is the piggybacked versions of DroidKungFu, Plankton, Kuguo, and Adwo.

2.3 Hardware Performance Counters (HPCs)

Hardware Performance Counters (HPCs) are considered to be special-purpose registers; developers might not be aware of the existence of HPCs unless they have specific objectives to achieve by using HPCs. HPCs are unique registers to profile the performance of hardware events in numeric form (i.e. the number of instructions or branches executed). The operation of HPCs is dependent on the sampling rate and duration; during the specific amount of time, HPCs count how many times the hardware events occurred within a certain interval [12]. Depending on the platform and the version, supported hardware events vary, but typically, branch, cache and instruction related events are common. Despite the unfamiliarity in utilization of HPCs, the benefits of using hardware components for monitoring performance are clear. In software, monitoring the performance of a target, such as a device or an application, and collecting the associated data were not as precise as what direct hardware components can provide: HPCs increase the accuracy of collected data.

The access to the HPCs can be performed via system calls by the Linux kernel. For example, PCs that use processors having the Performance Monitoring Unit (PMU) with Linux OS can access to HPCs through system calls, in particular using the application Perf. In Android smartphones based on the Linux kernel with ARM architecture having PMU, HPCs are accessible using the system call as well. In the case of Android platform, they provide the command-line CPU profiling tool called Simpleperf given in their Native Development Kit (NDK) package. NDK package is a toolset providing the native language support, such as C and C++. The profiling tool provides a total of 482 hardware-event parameters including some software events (e.g. alignment-faults, context-switches, etc).

Despite the numerous hardware events given by Simpleperf, there is the limitation of the number of performance counters available depending on a phone's specification; the accurate number of performance counters differs depending on the version of ARM architecture. Therefore, a few hardware events from the event list were selected. Based on the specification or a version of a processor on a device, some of the 482 events are not supported. However, generally four to six counters exist in a device and the device can monitor a number of parameters equal to the number of counters; this occurs simultaneously with full dedication of each performance counter. If more than four parameters are utilized where the device has four counters, each counter shares itself across the hardware events in a certain proportion. For instance, ARM Cortex-A5 supports only two performance counters, which means two HPCs can monitor the hardware events. ARM Cortex-A7 has 4 performance counters [13], and ARM Cortex-A9, A53, and A57 have 6 counters [14]. ARM cortex technical reference manual presents this information on the official website. Therefore, acknowledgement of the version of ARM architecture is important for the usage of HPCs.

2.4 Machine Learning for Classification

Historically, classification tasks that have an extensive amount of data have traditionally been cumbersome for researchers; an opportunity to improve efficiency is present with the use of machine learning. To do so, many classifiers are available for use in either the algorithm format or as a tool, such as WEKA. The classifiers are divided into two groups, supervised and unsupervised classifiers. Unsupervised classifiers require unlabeled data and analyze the pattern of the data, such as discovering abnormal data, whereas supervised classifiers use pre-labeled data. For instance, the collected data is applicable with supervised classifiers since the data is categorized into benign and malicious under each hardware feature. In terms of classifiers, the most popular classifiers are Decision Tree, Naive Bayes Classifier, K-Nearest Neighbors (KNN), Support Vector Machines (SVM), and Artificial Neural Networks, where Decision Tree, Naive Bayes Classifier, KNN, and SVM belong to supervised classifiers, and Artificial Neural Network classifier belongs

to either supervised and unsupervised. The classifiers specifically related to malware detection are the following: [15] used Decision Tree, KNN, and Regression, [2, 16] used ensemble classifier trained with KNN and one of Decision Tree classifiers, [17] used the SVM classifier. Based on [18], they studied classifiers used in previous work relevant to malware detection and concluded that the accuracy rate of malware detection is higher using the Random Forest Tree classifier, one of the Decision Tree classifiers, as compared to the SVM and Naive Bayesian classifiers. Based on the above research, OneR from rule-based classifiers, J48, RandomForest, and RandomTree from Decision Tree-based classifiers, KNN, and an ensemble classifier trained with KNN and Random Forest Tree were chosen. OneR classifier was developed by Rob Holte in 1993, and the concept is that one attribute does all the work. The classifier calculates the lowest error rate predictor using the formula: $TP + FP/T + F$, where TP is the true positive rate, and FP is the false positive rate. Then, the attribute with the lowest error rate classifies the data. The terms, such as TP and FP, are derived from the confusion matrix in machine learning, which will be discussed in Chapter 4. OneR is best fit for a simple dataset, as well as small, noisy, and complex datasets, which proves the simple way sometimes works the best. The J48 classifier selects good attributes for the root nodes by choosing the purest nodes, the greatest information gain based on the information theory: $Entropy(p_1, p_2, \ldots, p_n) = -p_1 log p_1 - p_2 log p_2 \ldots - p_n log p_n$. This formula is to measure the information in bits and the highest bits are selected as root nodes consecutively. It is important to note that one significant drawback for the tree-based classifiers is that they consume considerable memory while pruning the tree.

2.5 Malware Visualization Technique

Visualization concentrates on exemplifying malware features in a graphical form. For static analysis, visualization techniques have been actively used in a range from typical bar, dot, and pie graphs to Treemap, Thread Graph, and Linked Graph [19]. The Treemap technique is to describe the behavioral events in nested rectangles, while the Thread Graph technique may be considered the advanced version of Treemap where it presents

the behavioral features chronologically. The Linked Graph technique is used to present the hierarchy or the relationship of the network using nodes and edges. One example of a Linked Graph, a call graph, is generated with only an .apk file without execution and is widely chosen as a major static analysis method in malware detection. The call graph is composed of the starting points of an application pruning vertices and edges representing methods and functions that are called by callers. The vertices include the information whether an attribute is defined in a DEX file or APIs that are not defined in the DEX file. [20] used the call graph to extract the structural features for use of machine learning input as a training set and fed the training set to the deep graph convolution network. [21] extracted the information of the call graphs and implanted the graphs into a low dimensional feature vector to train Deep Neural Networks (DNNs), using similarity detection for training and testing in machine learning and discovering whether a target is malware or benign. In this thesis, the call graph given as an API by Androguard is utilized to show the visual difference of static analysis between the benign and malicious versions of the same applications to strengthen the effectiveness of HPCs, even in tough obfuscations at an application level.

2.6 Related Work

To conduct malicious update studies, many researchers use either network or permission-based research to detect updating malware due to the update characteristics; as many as three patents related to application updates using networks were released in 2020 [22–24] In one of the studies, researchers analyze the abnormality in bytes of packets incoming and outgoing via the network [7, 25]. In another study, they translated Java Bytecode of .class files to formal models of Calculus of Communication Systems (CCS) process specification, and used Mu-calculus logic to get the logic semantics, then arose the Concurrency Workbench of the New Century (CWB-NC) model that determines whether malware exists in an application [3]. The last case is that permission-based method is used to devise an attack [7] as well as detecting the update attack [26]. [2] adopted a recursive and self-learning process in their application of machine learning, and [27] developed

a command-line based Python tool that reproduces the repackaging that malignant developers do. [16] used voting classifier trained with KNN and Random Forest with static features by extracting them from the AndroidManifest.xml components, such as permissions, API calls, and dynamic features by using the random UI-manipulation tool Droidutan, which gives UI elements of activities.

2.6.1 Network pattern-based detection

[7, 25] targets the self-updating malware based on the analysis of network behavior observing the abnormal network behavior at the application-level. The process consists of feature extraction, feature aggregation, local learner, and anomaly detection. Feature extraction collects the data in a certain amount of time based on the features, such as sent/received bytes, network state, or application states whether running in the background or foreground. Then, the authors filtered features that can be useful in the machine learning phase: feature aggregation. During the local learner phase, the network pattern on each application was learned using the C4.5 Decision Tree algorithm to find any abnormal behavior in the anomaly detection phase. Also, the two classifiers, Decision Table and Decision/Regression tree (REPTree), measured five different levels of anomaly acceptance rate (i.e. 5%, 10%, 15%, 20%, 25%) on multiple network patterns from the different and same versions of applications, discovering up to what percentage the abnormal events are accepted in the regular applications. The anomaly acceptance rate resulted in around 20% and 25% within network patterns from the same version of applications, whereas a few of the patterns from different versions of applications behaved somewhat unpredictable. Therefore, they brought an alarm strategy, for example, if abnormal instances are detected three times consecutively, the pattern is considered to be a true warning. With the premise, classifying different versions of the same application and detecting self-updating malware with their system were evaluated with Decision Table and Decision/Regression tree (REPTree). For the first evaluation, detection accuracy achieved 87% with 20% of anomaly acceptance rate on Decision Table algorithm, whereas REPTree algorithm reached 94% accuracy with 25% anomaly acceptance rate. For the self-updating malware detection evaluation with their system, one of the applications

that contain update malware that they obtained from the application market and one of self-updating malware applications reached a considerably low true positive rate (i.e. 67.9% and 45%) even though those applications are considered to be less obscured with malicious payload. They discussed the reason why it obtained such a low result is that the main functionality of the application remains in the infected one. Other than those cases, mostly from 90% to 100% true positive rate and from 0% to 10% false positive rate are achieved.

2.6.2 Feedback-loop depicting active learning (Aion)

The author of [2] considered the detection of repackaged applications to be a search problem, because the execution path towards the malicious payload could be veiled profoundly and never occur during the examination. Aleieldin Salem devised an architecture and a platform having a process of stimulation, analysis, and detection with active learning and evaluating environment. In the active learning phase, a classifier categorizes an execution path, called a feature vector. If the classification is inaccurate, the classifier picks another feature vector and this process is repeated until the maximum accuracy of classification is achieved. In specific, the architecture consists of two parts: data generation and data inference. In other words, data generation is for adding new applications, and the data inference is for increasing the accuracy rate of the classifiers. In the data generation phase, collecting applications, analyzing the applications under the control to accumulate runtime behavior, recording the behavior, and converting the data to a comprehensive format, such as separating the texts with a delimiter, occur in a sequence. During the data inference phase, it tracks down the API calls to extract numeric data (i.e. counts of different API calls), eliminates noise in the data, extracts patterns and information, then trains a classifier, validates the results with a test dataset, and reports the result. The two phases are repeated until it reaches the highest accuracy. Furthermore, each phase can feed the result themselves. As a result with piggybacked applications, KNN with $k = 500$ attained lowest score and Random Forest was the highest scoring classifier, where this aspect is shown in my experiment as well. Based on the experiment of the architecture, the author concluded the following: firstly, either static

features themselves or dynamic features themselves are not sufficient for training and testing. Secondly, the highest F-measure value is dependent on the classifier and the feature type. Lastly, in spite of the feature type, some classifiers performed consistently.

Chapter 3
Experiment

3.1 Testing Environment Setup

A Samsung Galaxy S4 (locked to AT&T) smartphone was used to proceed with the experiment. The CPU comprises ARM-Cortex-A15 and A7, where A7 has four performance counters. The phone was rooted with KingoRoot to be capable of reading the performance counters via Simpleperf. The operating system of the Galaxy S4 is Android 5.0 (Lollipop) and was released in 2013 as described in Figure 3.1. The details of the phone specification could affect the further experiment due to the Simpleperf execution requirements and root. From Android version 5.0+, only a Simpleperf executable file can be run, and from version 7.0+, source code given in the Git repository will work since a device with the corresponding version includes the dependencies to build the source code. Executable Simpleperf is under the NDK in the Android library and different versions of NDK are available under the SDK tool in the SDK manager of Android Studio. The 20.1.5948944 version NDK for Galaxy S4 was downloaded to obtain the executable Simpleperf file under `/ndk_version/simpleperf/bin/android/corresponding_arm_architecture`. The NDK version listed in Figure 3.1 or below was executed without any noticeable hindrance. According to the default NDK version fixed by the Gradle plugin version, version 21 NDK should be compatible with Galaxy S4, however, it caused the illegal instructions error message. It means that the cross-compile has failed. However, it was the correct architecture, and the file was given by the Android

Device Specification	
Product Name	Samsung Galaxy S4
CPU	ARM Cortex A15 ARM Cortex A7
# of HPCs	4
Android Version	5.0.1 (Lollipop)
Compatible NDK Version	20.1.5948944

Figure 3.1: Device specification used for the experiment

official resource; cross-compile is not the reason. As an alternative, the one version below was selected and was successful. To check whether a version works on a smartphone, downloading the version of simpleperf to the phone via ADB (Android Debugging Bridge) shell, the command-line tool allowing users communication with an Android smartphone, is required. Subsequently, either an error message or the instructions of how to use Simpleperf is displayed. If error messages occurred, repetition to find a correct version of NDK is needed. The way to root the device varies based on the manufacturer, carrier, and the model number. KingoRoot has the list of devices compatible with their rooting tool. Lastly, the sandbox is the most important item in this experiment. Since the applications containing malware should be running on a phone, it is possible for the device to be infected. Fortunately, Android phones are built with Linux-SE meaning that all the applications run on a sandbox environment and individual applications do not interact with other applications; it protects other applications or data from being accessed by a malicious application. Thus, the environment setup is completed.

3.2 Simpleperf

Simpleperf is a native profiling tool provided in graphical interface and command-line form, and it supports the same commands as the Linux perf [28]. Performance Monitoring Units (PMUs) have specific registers where the hardware event parameters are assigned and counting is started and read by users. Linux perf supports about 600-700 hardware events and Simpleperf supports about 500. However, only a limited number of events can be profiled at the same time depending on the number of HPCs on a device.

Therefore, choice of hardware events are important since running all the hardware events is time-consuming and some events are not necessary to profile or not supported by a device.

Simpleperf Hardware Event List			
1. branch-load-misses	2. branch-loads	3. dTLB-loads	4. dTLB-stores
5. iTLB-loads	6. iTLB-stores	7. L1-dcache-load-misses	8. L1-dcache-store-misses
9. L1-dcache-stores	10. L1-icache-load-misses	11. L1-icache-store-misses	12. node-loads
13. node-stores	14. branch-instructions	15. branch-misses	16. instructions

Figure 3.2: Simpleperf hardware event list applied to the malware detection.

Figure 3.2 includes the effective hardware event parameters extracted with a feature reduction algorithm and the correlation of the features for classification [5]. In Figure 3.2, the events 1 through 13 are categorized to the hardware events and 14 through 16 are classified to the software events. The events in grey colored cells and blue colored cells are what the [5] refined with the algorithm. The parameters outlined in orange is what was picked based on what Hossein, Nisarg, Et al selected and [29] describing that the cache events are leveraged for malicious attacks, which implies that cache events can be used for detection of the attacks. The event dTLB-loads was chosen because the event is relevant to dTLB-stores. If so, either one of load or store can be used instead of using both. The yellow colored cell, instructions, is paramount because the nature of injecting malware, the piggybacking technique, is adding additional instructions in the existing application.

3.3 Dataset

Downloading piggybacked applications is more complicated than downloading regular malicious applications. For example, a framework, called HookRanker, for locating a malicious payload in the existing code was introduced in [8], but the program is no longer available. [4] presented the dataset called Malgenome containing more than 1200 samples of malware from August 2010 to October 2011, which is outdated. [30] lists the links connecting to malware datasets, such as Kharon, Drebin [17], and AMD [16] introduced in previous work. However, the Kharon dataset is significantly limited by having only DroidKungFu available, AMD official webpage is not accessible, and

Drebin dataset has about 5,560 applications available with 179 different malware families, collected from August 2010 to October 2012. UC Davis is one of the institutions that was given access, however, they were unreachable as well. Fortunately, AndroZoo [31], having approximately 14,832,925 different APKs where a few of them were transpired as malware, was available. To access their dataset, they require us to request the API key. They provide the infected apk files with a SHA256 key obtained by contacting them and the list of the available malware types and names on their website. Obtaining apk files is performed with the following command from the browser: *https://androzoo.uni.lu/api/download?apikey=***${APIKEY}***&sha256=***${SHA256}**, where bold texts are replaced with the corresponding key values. Our target is piggybacked applications meaning that a benign version and malicious version of the same application is required. The list of SHA256 keys given in the list of the types and names by AndroZoo is completely the list of malware only. Therefore, SHA256 keys matching to the same benign application are required, and Github resource [32] managed by the authors of [8] provides the SHA256 key of the original benign version of the applications matching to the malicious SHA256 keys in the AndroZoo list.

3.4 Automation of Experiment Pipeline

3.4.1 Download APK files from AndroZoo

When downloading piggybacked applications, a pipeline was devised to increase the efficiency of the experiment. The first step is to decide which name of the malware to download from the AndroZoo list and input in a command. Next, it finds all SHA256 keys having the corresponding malware name. Then, it searches for the benign version from the list given by the Github resource [32]. If the key exists in the list, the original benign version and malicious version of the same application can be downloaded. Over 1,302,971 keys for malware .apk files are available according to the list of malware name or type label given by AndroZoo and 1,498 keys, provided by [32], which are original applications, matching the piggybacked application keys, exist. This overwhelming process is automated in the framework and the framework expects a one line of the

command: *python3 main.py –malware* {*malware_name*} *–save_dir* {*/path/to/save/apk*} *–download*. This command creates folders named with the number of applications downloaded in an increasing order under the specified by the user, followed with a benign and malicious folder, and an .apk file or multiple .apk files on the existence of multiple versions in each folder.

3.4.2 Collect Data with HPCs

The process of collecting the HPCs monitoring data is quite complicated. This data is based on Samsung Galaxy S4, details are described in section 3.1 in Chapter 3. As we can observe in Figure 3.3, each step was accomplished manually at the beginning of the data collection and consumed considerable time and full attentiveness so that the next step may proceed; the importance of automation of the process became apparent. The overview of the process is that one application runs 80 cycles for a benign application and piggybacked application, respectively. Since the smartphone that was used for the research has four performance counters, each row in Figure 3.2 consists of a *event_group* in Figure 3.3. Each *event_group* performs 20 times, which is presented as *bound* in Figure 3.3, in 80 cycles.

The automated bash script loads the proper version of Simpleperf to a target smartphone via Android Debug Bridge (ADB). Therefore, the first step for a user is to acknowledge the accurate Simpleperf version for the bash script. A way to find the corresponding version is mentioned in section 3.1 of Chapter 3. Then, the benign application is installed before the malicious one because a factory reset needs to be executed for the security purpose and isolation of the testing applications, which carries extra time consumed in factory reset and basic essential setup by hand to return to the home screen. The benign .apk file is installed through ADB, and the package name should be given as an input as it is necessary for the next procedure. Depending on where the dataset is obtained, the .apk file name might hold a package name, a SHA key, or other formats. In my case, .apk filenames are composed of a SHA key, therefore, use of another tool was inevitable to extract the package name. The tool chosen is called Androguard because of the modulability, stability and portability for integration in the framework, though other similar programs are readily available for download online.

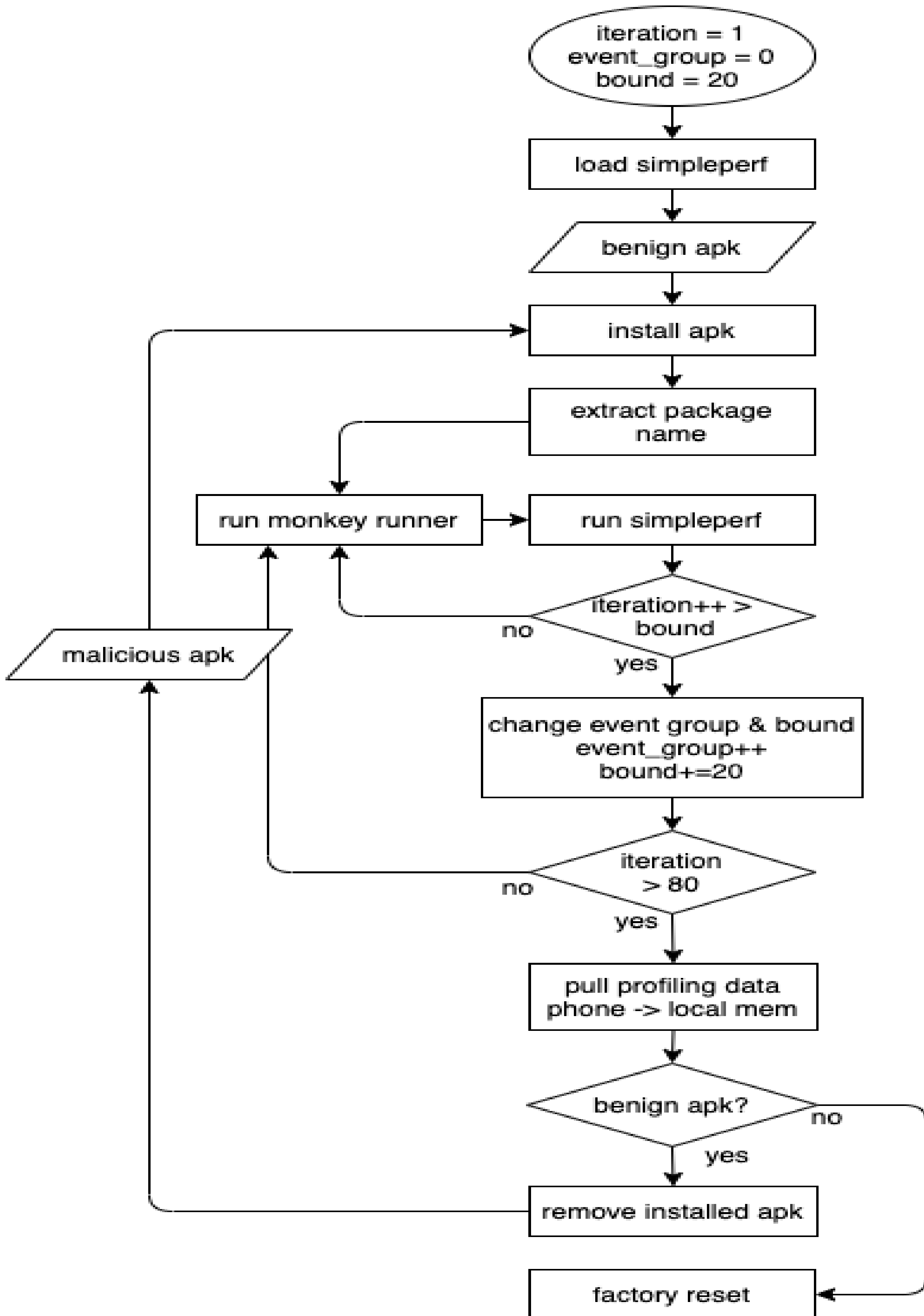

Figure 3.3: Automated process of collecting the HPCs Monitoring Data

A few functionalities from Androguard are integrated in the framework; obtaining the package name within the framework is available. With the package name as input to the monkeyrunner phase, the target application will be run automatically. Monkeyrunner is a tool that maneuvers an Android device at a functional or framework level. The tool provides special features: multiple device control, functional testing, regression testing, and extensible automation. The functional activities of monkeyrunner include sending keystrokes or touch events, exploring the menus, and taking screenshots. Once monkeyrunner is initiated, the Android device creates the process id of the running application. This explains why Simpleperf must operate after monkeyrunner. The command of Simpleperf is in the following format: *./simpleperf stat -p {process_id} -e {event_group} --duration α --interval 10 -o output.csv*. The script reads the process id pulled with the package name. Duration α is set to 15s because monkeyrunner runs approximately 5s at the shortest and 13s at the longest. 15s should be long enough to not lose the testing results. Sampling rate is at 10ms, which is the lowest interval the tool allows due to the unavoidable overhead of the profiling platform. Event group is covered beforehand, thus the inputs for Simpleperf are ready. While monkeyrunner and Simpleperf are repeated for 80 cycles, every 20 cycles, the script replaces the event group to profile. When the iteration reaches 80, output files saved in a designated directory in the phone are pulled out to the *{DESKTOP_PATH}/benign_output* folder. After the results are moved completely from the smartphone to the location mentioned above, the script uninstalls the downloaded .apk file, and installs a malicious .apk file. The pipeline explained above is repeated for the malicious version, and once malicious version is completed, the script triggers the factory reset.

3.4.3 Transformation of the raw data in understandable format

After collecting the data with HPCs, unrefined raw outputs are segregated into multiple files. Figure 3.4 shows a part of a file having four hardware event parameters out of 16 parameters: branch-load-misses, branch-loads, dTLB-loads, and dTLB-stores. These data need to be transformed into a comprehensible format for the application of machine learning. The necessary process is to gather the separated hardware events, as well as

820	648 branch-load-misses # 97.692818% miss rate (100%)	
840	029 branch-loads # (100%)	
1	760	247 dTLB-loads # (100%)
1	854	218 dTLB-stores # (100%)
Total test time: 0.810883 seconds.		
Performance counter statistics:		
2	749	657 branch-load-misses # 98.534837% miss
2	790	543 branch-loads # (100%)
6	853	115 dTLB-loads # (100%)
6	915	270 dTLB-stores # (100%)

Figure 3.4: Raw output of Hardware Performance Counters

branch-load-misses	branch-loads	dTLB-loads	dTLB-stores	iTLB-loads	iTLB-stores	L1-dcache-load-misses	L1-dcache-store-misses	L1-dcache-stores	L1-icache-load-misses	L1-icache-store-misses	node-loads	node-stores	branch-instructions	branch-misses	instructions	type	apk	apk-match
																benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
																benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
																benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
																benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
																benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
																benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
																benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
																benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
																benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
																benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
																benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
																benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
																benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
																benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
5104213.0	5104213.0	14233432.0	14233432.0										138659.0	13845.0	663026.0	benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
7953135.0	7953135.0	21539539.0	21539539.0	210545.0	210545.0	155754.0							2165900.0	267577.0	14838652.0	benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
11398568.0	11398568.0	30225602.0	30225602.0	503802.0	504264.0	467556.0							4903501.0	576366.0	36853320.0	benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
17432422.0	17432422.0	45322670.0	45322670.0	754111.0	755231.0	719508.0							7436840.0	1262058.0	58223652.0	benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
21709917.0	21790655.0	56374873.0	56374873.0	1177317.0	1177317.0	1209443.0							11191407.0	2325718.0	89397105.0	benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
24879983.0	24879983.0	61663316.0	61663316.0	1403565.0	1403565.0	1358273.0							15097011.0	3302778.0	120602389.0	benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
27066263.0	27066263.0	65161723.0	65161723.0	1603699.0	1603699.0	1497014.0							18149691.0	4227185.0	145676184.0	benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
31808227.0	31808227.0	74950330.0	74950330.0	2459387.0	2462600.0	2082034.0							21705698.0	4612198.0	168994214.0	benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
32490039.0	32490039.0	76712512.0	76712512.0	3220165.0	3220749.0	2552904.0							24675064.0	4902530.0	188917257.0	benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
41894947.0	41894947.0	99831432.0	99831432.0	3678233.0	3681182.0	2861345.0							28297820.0	5644594.0	215483723.0	benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
44389949.0	44389949.0	105138543.0	105138543.0	4214376.0	4214376.0	3324613.0							32300362.0	6784476.0	248049562.0	benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
46327936.0	46327936.0	110876332.0	110876332.0	4402810.0	4403312.0	3459060.0							36011825.0	7524796.0	276416307.0	benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
47755178.0	47755178.0	113465072.0	113465072.0	4569223.0	4571394.0	3672504.0							38818012.0	7842600.0	297836287.0	benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
48066335.0	48066335.0	113938463.0	113938463.0	4972355.0	4972355.0	4032726.0							47235831.0	9174855.0	364404818.0	benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
48175325.0	48176390.0	114266829.0	114271877.0	5356466.0	5360195.0	4406364.0							49924754.0	9507255.0	386512378.0	benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk
53631869.0	53637708.0	125174577.0	125192715.0	6377600.0	6381307.0	5094385.0							51845332.0	9774579.0	400769613.0	benign	E6273AEE16E2CEA4F08526302D59768D9DFD420C281DA09FCFCB9190BB709EDE.apk	1CA20CC2CCF0E981781B25BAD314098A168B6FAF848393A37D19A302A40F3F4C.apk

Figure 3.5: Refined and optimized output of Hardware Performance Counters

all the performance counters values from the benign version and malicious version of the same application into one file. Therefore, the framework offers a script for the data refinement and optimization. It extracts the hardware event parameters and numbers, and congregates them. Specifically, in Figure 3.5, the top row lists 16 hardware event features, whether the data of each row belongs to the benign or malicious .apk file, .apk file names, and the name of the pair of the .apk file. Also, this process removes the zeros because the zero values occur while HPCs start counting when the applications are inactive based on the raw data.

3.5 Machine Learning

In this section, the comprehensive data from the previous step is fed to the classifiers for training and testing. The WEKA tool was utilized for classification and evaluation. As discussed from section 2.4 in Chapter 2, OneR, KNN, J48, Random Forest, Random

	Machine Learning	
Malware Type	**Training**	**Testing**
Piggybacked Adware	Adwo (70%) + benign version	Adwo (30%) and Kuguo + benign version
Piggybacked Trojan	DroidKungFu (70%) + benign version	DroidKungFu (30%) and Plankton + benign version

Figure 3.6: The plan of machine learning training and testing set in accordance with Trojan and Adware malware types

Tree, and Ensemble with KNN and Random Forest are the classifier candidates. The prepared dataset encompasses two different Adware categories and two different Trojan categories. Kuguo and Adwo are the former and DroidKungFu and Plankton are the latter. Firstly, Adwo and Kuguo (Adware) are examined to explore if HPCs are utilizable to distinguish piggybacked malicious applications that do not have update attack. If successful, one step further is proceeded, which is the objective of the research; whether HPCs are feasible to use in detection of the piggybacked applications with update attack, which are DroidKungFu and Plankton, is investigated. To do so, 70-75% of the data from one of each malware type, Adwo and DroidKungFu, and the benign versions of them are used for the classifier training while the remainder of the data is used for testing to explore how accurately the classifier is at detection as shown in Figure 3.6. The entire Kuguo and Plankton dataset along with the benign versions of them is solely used for testing and was not used to train any classifiers. This testing is for expanding the capability of HPCs detection scope within the same malware type instead of limiting to one in each category. It is important to note that WEKA tends to accumulate the training and testing results, therefore, the buffer has to be deleted after one set of training and testing per classifier.

Chapter 4
Discussion

4.1 Piggybacked applications

In this experiment, over 300 applications were downloaded across the four different malware names: Adwo, Kuguo, DroidKungFu, and Plankton. Adwo and Kuguo are Adware and the other two are Trojan. However, the applications that were runnable and data-collectable were very limited. The reason for this is that some applications are not executable due to existing bugs, unmatching Android OS version, or limitations confronted by monkeyrunner. Due to the characteristics of monkeyrunner, game applications requiring sophisticated control are not suitable for monkeyrunner to explore. Therefore, the data collected with user's manipulation were excluded for a fair comparison. As a result, 40 applications were collected. These applications are partitioned into four groups, which is again divided into two groups; benign version and malware Adwo, Kuguo, DroidKungFu, and Plankton versions, accordingly. Fourteen applications are included in the first group, where a half of them is benign version and the other half is malware Adwo. Each malicious version is originated from the corresponding benign application. In the same manner, 8 applications belong to second group, where a half is benign and the rest is Kuguo, 16 applications are in third group, and 2 applications are in the last group. They are entitled as the following: the first group as Adwo group, the second group as Kuguo group, the third as DroidKungFu group and lastly as Plankton group. In this section, we determine if a piggybacked application can be detected using HPCs. If successfully

detected, the end goal is to is to further apply HPCs to determine if an application has been infected with update malware. Firstly, the benign version and piggybacked version of Adwo and Kuguo applications are discussed to investigate how efficient HPCs are within the piggybacking technique. Then, the benign version and piggybacked version with update attack, which are DroidKungFu and Plankton, are observed with the data collected by HPCs.

4.1.1 Piggybacked Malware: Adwo and Kuguo (Adware type)

As mentioned above, this section covers the 14 Adwo group and 8 Kuguo group to discern if a piggybacked application can be distinguished using HPCs. Ten of the applications in the Adwo group were selected, half benign and half Adwo, to be used for training while the remaining four were used for testing to observe how correctly the classifiers categorize the same malware. Then, the training set was used to test 100% of the applications relevant to Kuguo to discover whether it is classified properly within the same malware type family. Of the samples collected for analysis, the piggybacked malware data instances

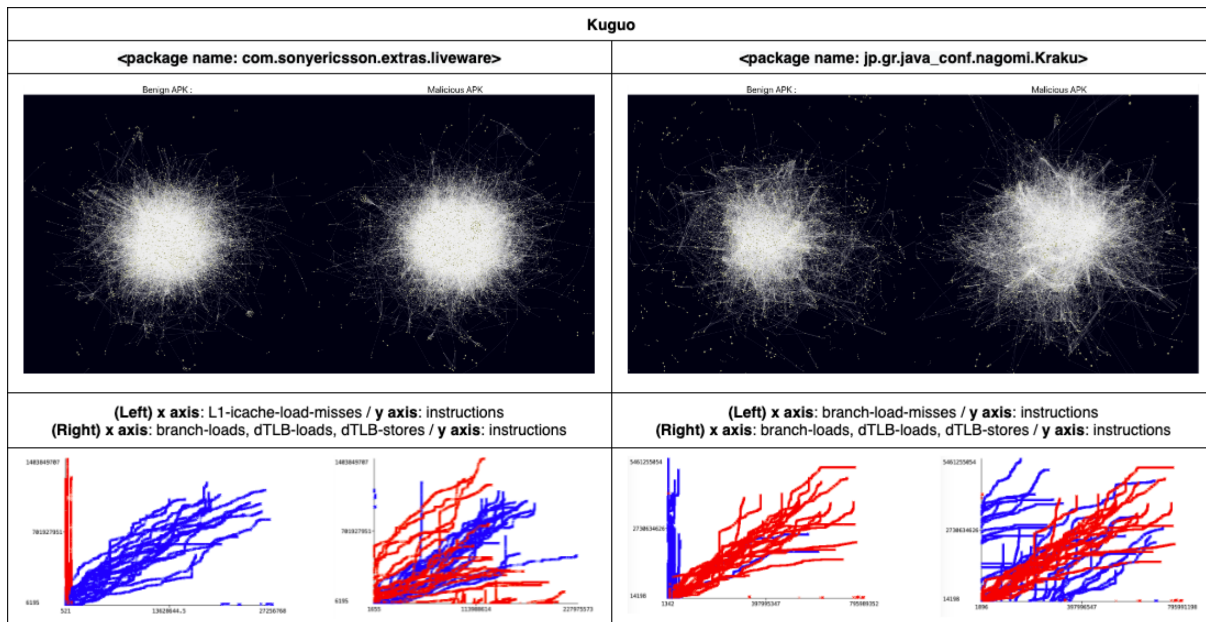

Figure 4.1: Obscure call graph with distinct HPC data pattern on Kuguo applications (Note: In the call graphs, the left shows benign version and the right displays malicious version. In the scatter plots, red represents malicious version and blue represents benign version.)

are somewhat different from the piggybacked update attack malware type; this will be discussed in further detail in the following section. The discrepancy between all of the benign applications and malicious applications named Adwo is obvious based on the call graph and the scatter graph per hardware event. Some of the Kuguo applications and the associated benign version have obscure patterns in the call graph, however, the scatter graphs display distinctive patterns as shown in Figure 4.1. After analyzing the static and dynamic data, it is evident HPCs should be capable of distinguishing the benign version and piggybacked version of a same application.

4.1.2 Piggybacked Update Malware: DroidKungFu and Plankton (Trojan type)

In this section, the piggybacked applications deploying the update attack are examined. As mentioned above, the 16 DroidKungFu group and 2 Plankton group are covered to discern if a piggybacked application can be distinguished using HPCs. Twelve of the applications in the DroidKungFu group were selected, half benign and half DroidKungFu, to be used for training while the remaining four were used for testing to observe how correctly the classifiers categorize the same malware. Then, the training set was used to test 100% of the applications relevant to Plankton to discover whether it is classified properly within the same malware type family.

Figure 4.2 a shows the bar graph per each hardware event on the four of the benign version and piggybacked DroidKungFu applications: Fileman (top left), ClockSync (top right), com.danxinben.xs (bottom left), and com.notebook (bottom right). The application in each cell in Figure 4.2 corresponds to the cell in Figure 4.3, which is the call graphs. As introduced about the call graph in Chapter 2, all the edges and vertices represent the execution paths, functions, and callers, which is a static based pattern. The image displays a benign and malicious version of the same application on the left and right in each cell. On the left column in Figure 4.3, it is distinct that a significant amount of malicious payloads were injected from the benign version, whereas it is obscure to define which one is malicious or benign on the right column. The same characteristic is observed in Figure 4.2, where blue bar represents the benign version and red bar is the malicious

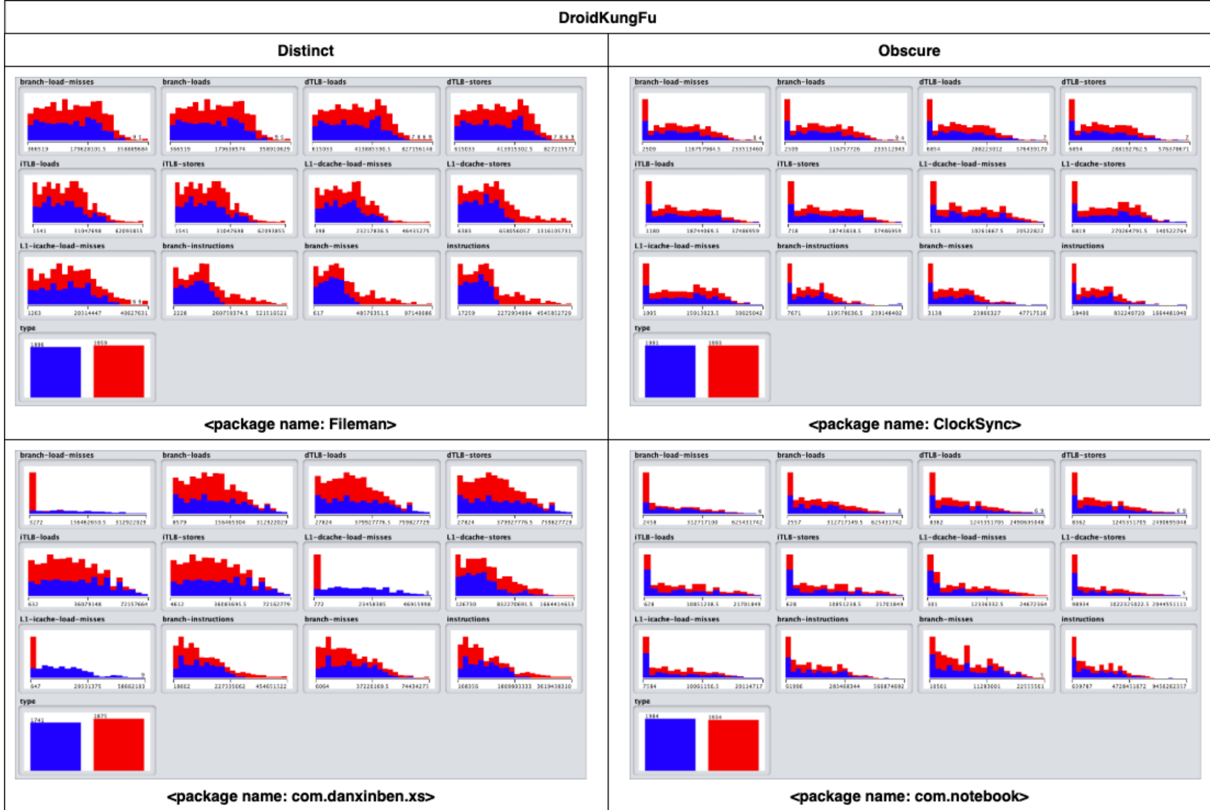

Figure 4.2: Bar graph of four training dataset of benign version and piggybacked DroidKungFu per hardware event by WEKA (Note: The bar plots in red represent malicious version, whereas the one in blue represents benign version.)

version. The Fileman application on the top left in Figure 4.2 clearly shows that a higher number of counts on every hardware event are perceived only in malicious versions, and the com.danxinben.xs application on the bottom left shows a higher number of counts on some events in the malicious version and other events in the benign version. In contrast, the counts for both benign and malicious versions of each application on the right column in Figure 4.2, in which each malicious version is obfuscated, are spread across the overall range.

Figure 4.4 represents the visualized HPC data on the hardware events with the Fileman and ClockSync applications that were on the top left and the top right of Figure 4.2 and Figure 4.3. Regardless of applications, some of the hardware events were able to be grouped with similar patterns, which means reducing the number of hardware parameters is possible: *(branch-loads, dTLB-loads, dTLB-stores), (iTLB-loads, iTLB-stores), and*

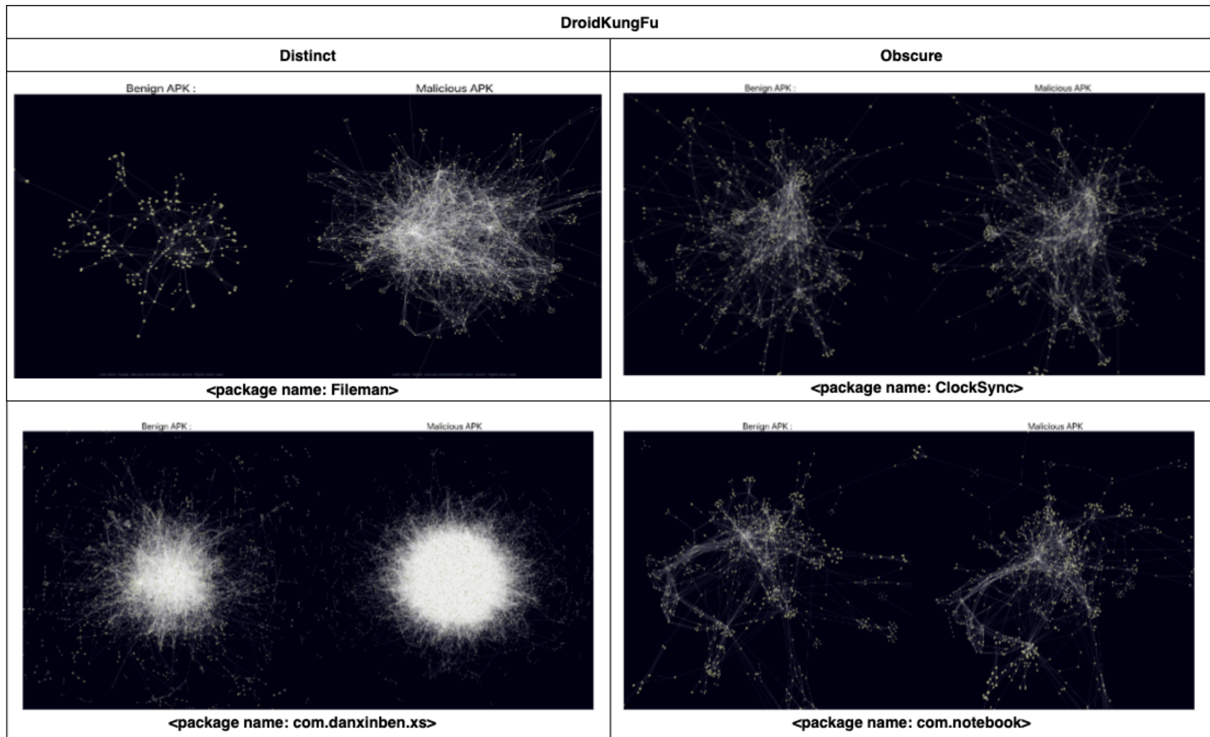

Figure 4.3: call graph of four training dataset of benign version and piggybacked DroidKungFu (Note: The left side of each call graph displays benign version and the right side displays malicious version.)

(branch-misses, branch-instructions). Since the number of HPCs is limited, minimizing the parameters is significant. All the other events in Figure 3.2 sometimes resulted in a similar pattern of those grouped instances and can be combined, but it varies depending on applications. In this case, the pattern for the hardware events grouped into four for both of the applications: *(branch-load-misses, branch-loads, dTLB-loads, dTLB-stores), (branch-misses, branch-instructions), (iTLB-loads, iTLB-stores, L1-dcache-load-misses), (L1-dcache-stores, L1-icache-load-misses)*. As we can see in Figure 4.4, Fileman draws a distinctive pattern; it should be easier for classifiers to classify the applications that have such a pattern. On the other hand, the pattern of ClockSync is vague with human eyes, yet the characteristic of the pattern must exist according to the evaluation result. For instance, the scatter graph pattern in the malicious version (red) of the ClockSync application is more congested whereas the benign version (blue) of the application is more dispersed in the first pattern graph.

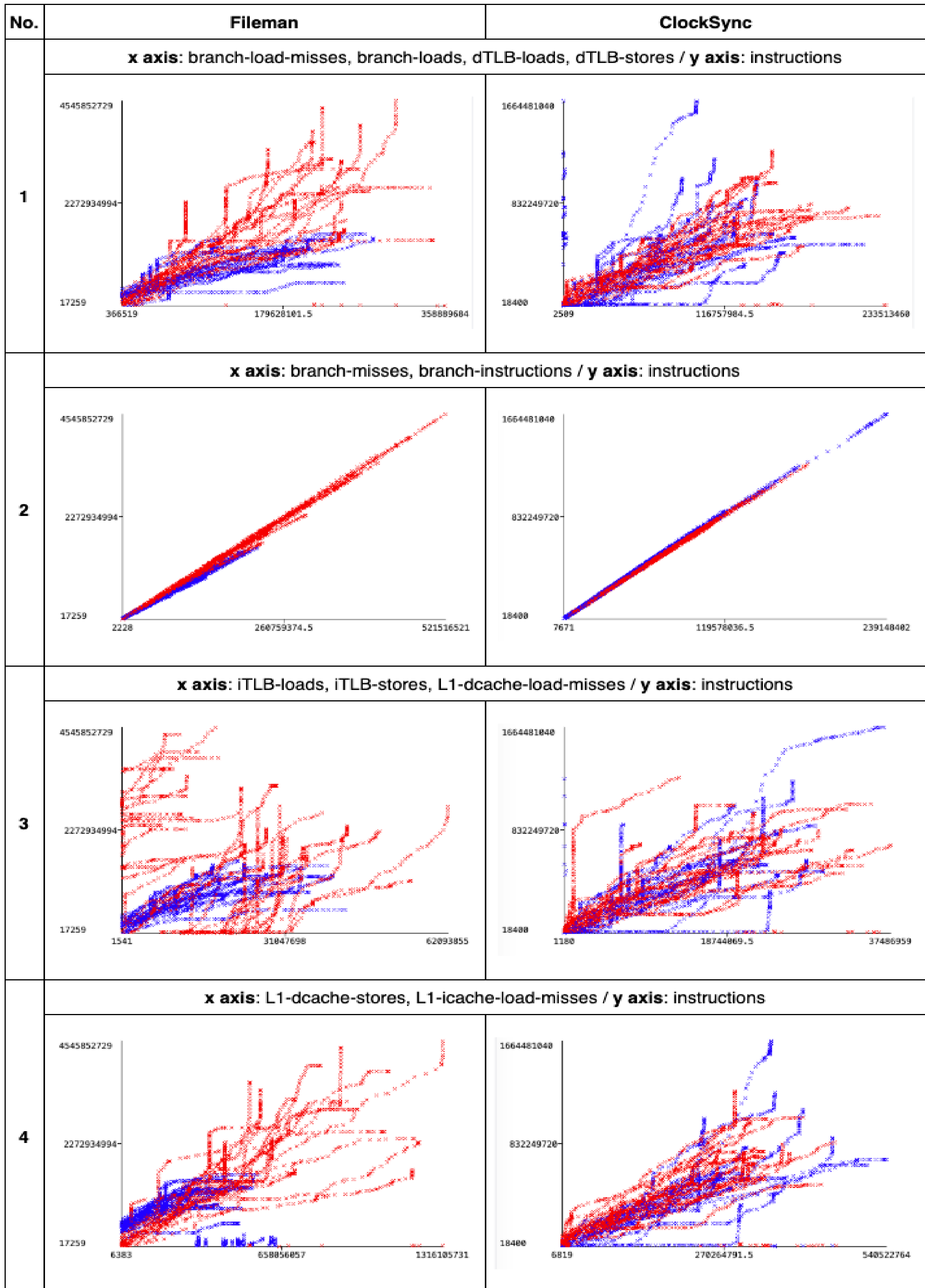

Figure 4.4: Scatter pattern graph of groups of hardware events on Fileman and ClockSync applications (Note: Red represents malicious version and blue represents benign version.)

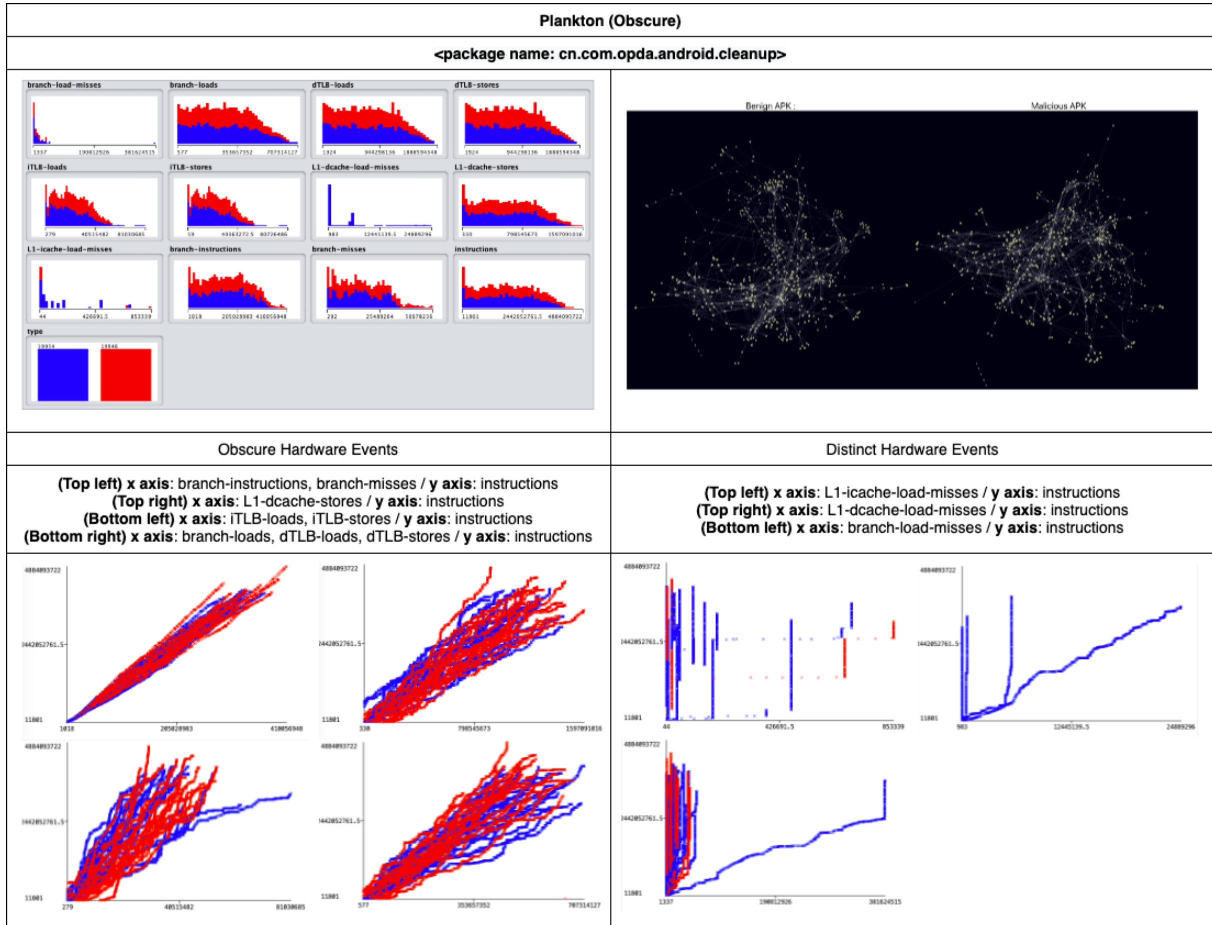

Figure 4.5: Visualized data for Plankton application used for testing (Note: In the call graphs, the left shows benign version and the right displays malicious version. In the scatter and bar plots, red represents malicious version and blue represents benign version.)

The Plankton applications and the associated benign versions, used for testing, were indistinct with static data shown in the call graph, whereas a small number of somewhat distinctive features in dynamic analysis, especially on the bottom right of Figure 4.5, exists. The hardware events that display discrete patterns are L1-icache-load-misses, L1-dcache-load-misses, and branch-load-misses. However, there is a concern that should be addressed. Although the patterns in the DroidKungFu group do not seem to have similarity with the Plankton group, which might cause the low detection accuracy at the first glance, there is some data demonstrating the analogous patterns leading to the high accuracy in classification of Plankton instances as like Figure 4.6; L1-dcache-load-misses data pattern from Plankton in Figure 4.5 shows some similarity in the pattern of the same

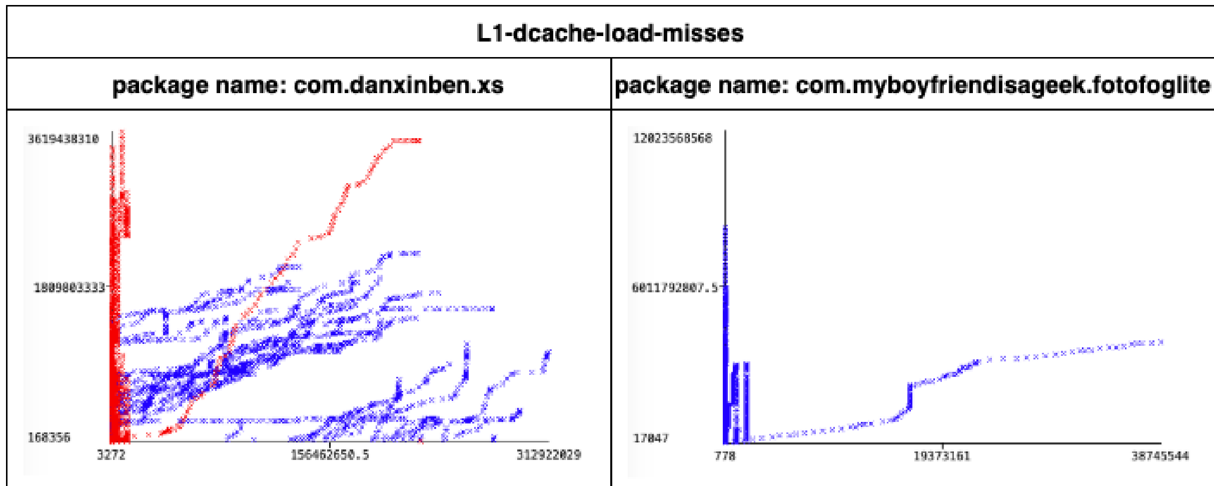

Figure 4.6: One of DroidKungFu Training data pattern on L1-dcache-load-misses close to Plankton testing data (Note: Red represents malicious version and blue represents benign version.)

hardware event in Figure 4.6.

4.1.3 Discovery

Based on the discussion, a few things were discovered. First, from the figures in this chapter, it is acknowledged that three types of different static and dynamic configurations exist, proving that utilizing only one or the other is inadequate to detect update malware. One type is a big-difference-existing call graph with clearly different performance counter data patterns. This type is seen in both of the malware types, Adware and Trojan. Another type is a barely-difference-existing call graph with indistinguishable patterns of the counter data. This type was observed by update malware (Trojan type) injected with the piggybacking technique. The last type is a barely-difference-existing call graph with clear patterns of the HPC data, which was perceived by some of piggybacked non-update attack malware (Adware type). Second, most of the call graphs, static information, correspond to the graph, where branch-instructions or branch-misses on x-axis and instructions on y-axis. The reason why it is *most* of the call graphs is that one out of the total collection had a large amount of malicious payloads added, however, the scatter dots appeared hugely in the benign version on branch-instructions and branch-misses hardware events instead of the malicious version. In this case, it

is assumed that malevolent developers added code without using condition variables, which is an explicit way of adding the malicious payload, and the malicious behavior was triggered immediately during the dynamic analysis. Lastly, a few hardware events patterns, overlapped in both of the malware types, are observed so that the seven events mentioned beforehand can be reduced to three.

4.2 Evaluation

In this section, the benign version and malicious version applications in non-update version piggybacked malware and update version piggybacked malware are evaluated. The former is the Adware type malware, and the latter is the Trojan type malware. To be more specific, Adwo and Kuguo are chosen for Adware, and DroidKungFu and Plankton are chosen for Trojan. The classifiers trained with 70% of the HPC data from the benign version and malicious version applications in Adwo, and tested with the remainder of the Adwo data and associated benign data, in addition to all of the data collected with the Kuguo group, which is in the same Adware malware type, were evaluated. To be specific, the data collected from 10 applications in the Adwo group, half benign and half malicious of the same applications, are used for training the classifiers, and the rest is used for testing. Then, whether the data, associated with the Kuguo group in the same malware family (Adware), is detectable with the classifiers that were used above, was explored. This allows me to evaluate the usability of HPCs against piggybacked applications and observe how correspondingly the classifiers categorize them within the same malware category into malicious and benign. It resulted in that use of HPCs in piggybacked applications is possible. Thus, the piggybacked applications containing update attack are used to evaluate whether the HPCs are utilizable in update malware detection. Therefore, similarly, the classifiers were trained with 70% of the HPC data from the benign version and malicious version of DroidKungFu applications, and tested with the remainder of DroidKungFu data and associated benign data and entire dataset affiliated with the Plankton group. In detail, the data collected from 12 applications in the DroidKungFu group, half benign and half malicious of the same applications, are employed for training

purposes, and testing was performed with the rest of the DroidKungFu group instances. The data gathered by the benign and malicious version of Plankton applications are tested with the classifiers that were used above to discover whether the data, affiliated with Plankton, in the same Trojan type, is discernible. If the detection rate of Adwo to Kuguo and DroidKungFu to Plankton is not satisfactory, adding a portion of Kuguo and Plankton to its training set was considered, but not necessary. The choice of classifiers are J48, RandomForest, RandomTree, OneR, KNN (k=3), Ensemble trained with KNN and RandomForest. The matrices adopted for the evaluation are the following:

- **Detect accuracy:** defined as the percentage of how accurately all the instances of the dataset was predicted and classified. This measurement is the most intuitive.

		Predicted Class	
		Yes	No
Actual Class	Yes	True Positive (TP)	False Negative (FN)
	No	False Positive (FP)	True Negative (TN)

Figure 4.7: Machine learning confusion matrix

- **True Positive Rate (TP rate), False Positive Rate (FP rate):** TP rate formula is TP/(TP+FN) and FP rate formula is FP/(FP+TN). Figure 4.7 describes all the abbreviations.

- **Precision:** defined as the percentage of correctly classified instances over total instances classified correctly. In other words, in the instances predicted to be malicious or benign, how many instances are actually malicious or benign. The formula follows TP/(TP+FP).

- **Recall:** defined as the percentage of correctly classified predictions over the truly classified and the ones classified wrong. In other words, in the instances correctly labeled to benign or malicious, how many instances belong to what it is labeled. The formula for recall is TP/(TP+FN).

- **F-Measure (F1-Score):** the weighted average of Precision and Recall. This is less intuitive compared to Detect accuracy, however, this F1 score is proper data to look

at when FP and FN are considerably different. The higher the score is, the better a classifier is. The equation for F-Measure is (2recallprecision)/(recall+precision)

- **ROC Area:** the value of ROC is based on the ROC curve graph. The more curvy the graph is, the higher the value that can be obtained. Being close to 1 means the classification works outstanding. In contrast, if the value is 0.5, this means no discrimination, such as fliping a coin.

- **Latency:** time taken to test the testing dataset with the trained classifiers.

Type	Classifiers	Precision		Recall		F-Measure		True Positive Rate		False Positive Rate		ROC Area		Training Set	Testing Set
		benign	malicious	benign	malicious	benign	malicious	benign	malicious	benign	malicious	benign	malicious		
Rule	OneR	0.889	0.875	0.921	0.828	0.905	0.851	0.921	0.828	0.172	0.079	0.874	0.874	Adwo (273413 instances)	Adwo (98767 instances)
Lazy	KNN	0.622	1	1	0.088	0.767	0.162	1	0.088	0.912	0	0.462	0.462		
Tree	J48	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.001	0.001	1	1		
	Random Tree	1	1	1	1	1	1	1	1	0	0	1	1		
	Random Forest	1	1	1	1	1	1	1	1	0	0	1	1		
Misc	Ensemble (KNN+Random Forest)	1	1	1	1	1	1	1	1	0	0	1	1		
Rule	OneR	0.885	0.842	0.83	0.894	0.857	0.867	0.83	0.894	0.106	0.17	0.862	0.862	Adwo (273413 instances)	Kuguo (138714 instances)
Lazy	KNN	0.806	0.713	0.658	0.843	0.724	0.773	0.658	0.843	0.157	0.342	0.752	0.752		
Tree	J48	0.999	0.971	0.97	0.999	0.984	0.985	0.97	0.999	0.001	0.03	0.999	0.999		
	Random Tree	1	0.989	0.989	1	0.994	0.994	0.989	1	0	0.011	1	1		
	Random Forest Kuguo group1	0.996	0.978	0.978	0.996	0.987	0.987	0.987	0.996	0.004	0.022	1	1		Kuguo (58012 out of 138714 instances)
	Random Forest Kuguo group2	1	0.998	0.997	1	0.999	0.999	0.997	1	0	0.003	1	1		Kuguo (39537 out of 138714 instances)
	Random Forest Kuguo group3	0.998	1	1	0.998	0.999	0.999	1	0.998	0.002	0	1	1		Kuguo (39175 out of 138714 instances)
	Random Forest Kuguo group4	1	0.986	0.986	1	0.993	0.993	0.986	1	0	0.014	1	1		Kuguo (1990 out of 138714 instances)
Misc	Ensemble (KNN+Random Forest) Kuguo group1	0.986	0.997	0.997	0.986	0.992	0.991	0.997	0.986	0.014	0.003	0.999	0.999		Kuguo (1990 out of 138714 instances)
	Ensemble (KNN+Random Forest) Kuguo group2	0.998	1	1	0.998	0.999	0.999	1	0.998	0.002	0	1	1		Kuguo (39537 out of 138714 instances)
	Ensemble (KNN+Random Forest) Kuguo group3	0.998	1	1	0.998	0.999	0.999	1	0.998	0.002	0	1	1		Kuguo (39537 out of 138714 instances)
	Ensemble (KNN+Random Forest) Kuguo group4	1	0.985	0.985	1	0.992	0.992	0.985	1	0	0.015	1	1		Kuguo (39175 out of 138714 instances)

Figure 4.8: Machine learning results on the classifiers based on the evaluation matrices (Adwo and Kuguo group)

Type	Classifiers	Precision		Recall		F-Measure		True Positive Rate		False Positive Rate		ROC Area		Training Set	Testing Set
		benign	malicious	benign	malicious	benign	malicious	benign	malicious	benign	malicious	benign	malicious		
Rule	OneR	0.857	0.838	0.832	0.862	0.845	0.85	0.832	0.862	0.138	0.168	0.847	0.847	DroidKungFu (22774 instances)	DroidKungFu (7770 instances)
Lazy	KNN	0.722	0.932	0.954	0.634	0.822	0.755	0.954	0.634	0.366	0.046	0.815	0.815		
Tree	J48	0.977	0.967	0.966	0.977	0.972	0.972	0.966	0.977	0.023	0.034	0.996	0.996		
	Random Tree	1	0.999	0.999	1	1	1	0.999	1	0	0.001	1	1		
	Random Forest	1	0.999	0.999	1	1	1	0.999	1	0	0.001	1	1		
Misc	Ensemble (KNN+Random Forest)	0.994	0.999	0.999	0.994	0.997	0.997	0.999	0.994	0.006	0.001	0.999	0.999		
Rule	OneR	0.988	0.924	0.919	0.989	0.952	0.956	0.919	0.989	0.011	0.081	0.954	0.954	DroidKungFu (22774 instances)	Plankton (39900 instances)
Lazy	KNN	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A		
Tree	J48	0.946	0.981	0.982	0.944	0.963	0.962	0.982	0.944	0.056	0.018	0.996	0.996		
	Random Tree	0.952	0.996	0.996	0.95	0.974	0.973	0.996	0.95	0.05	0.004	0.999	0.999		
	Random Forest	0.952	0.996	0.996	0.95	0.974	0.972	0.996	0.95	0.05	0.004	0.999	0.999		
Misc	Ensemble (KNN+Random Forest)	0.949	0.998	0.998	0.946	0.973	0.972	0.998	0.946	0.054	0.002	0.998	0.998		

Figure 4.9: Machine learning results on the classifiers based on the evaluation matrices (DroidKungFu and Plankton group)

Figure 4.8 describes the results from the Adwo and Kuguo group, where the malicious versions are the Adware type, and Figure 4.9 describes the results from the DroidKungFu and Plankton group, where the malicious versions are the Trojan type. Out of the listed matrices, the detection accuracy and latency were primarily discussed as seen in Figure 4.10, because latency helps to filter out the computation-heavy classifiers and the result of detection accuracy represents the Precision, Recall, and F1 score in the figures. Also, my goal is to find whether HPCs are adoptable in update malware detection. The figure attached above is additional information when necessary. In Figure 4.8, the results of Random Forest Tree and Ensemble were divided into four groups since the classifiers were not capable of displaying the result due to the size.

Considering the complexity of the instances and the previous work, it is expected that OneR and KNN (K=3) classifiers would not perform well. The KNN classifier trained with the dataset from the DroidKungFu group and tested with the dataset from the Plankton group did not provide the reportable result, which explains why the orange bar on KNN is empty in Figure 4.10. As covered in Chapter 2, OneR accomplishes classifications better with a small amount of data, as well as KNN. Additionally, some of benign and malicious versions of the testing dataset are not easily comparable due to the stealthiness; the combination of tough obfuscation and the simplicity of classifiers would have resulted in a higher false positive rate in those two classifiers. [18] studied the classifiers used for malware detection and came to the conclusion that Tree type classifiers, especially the

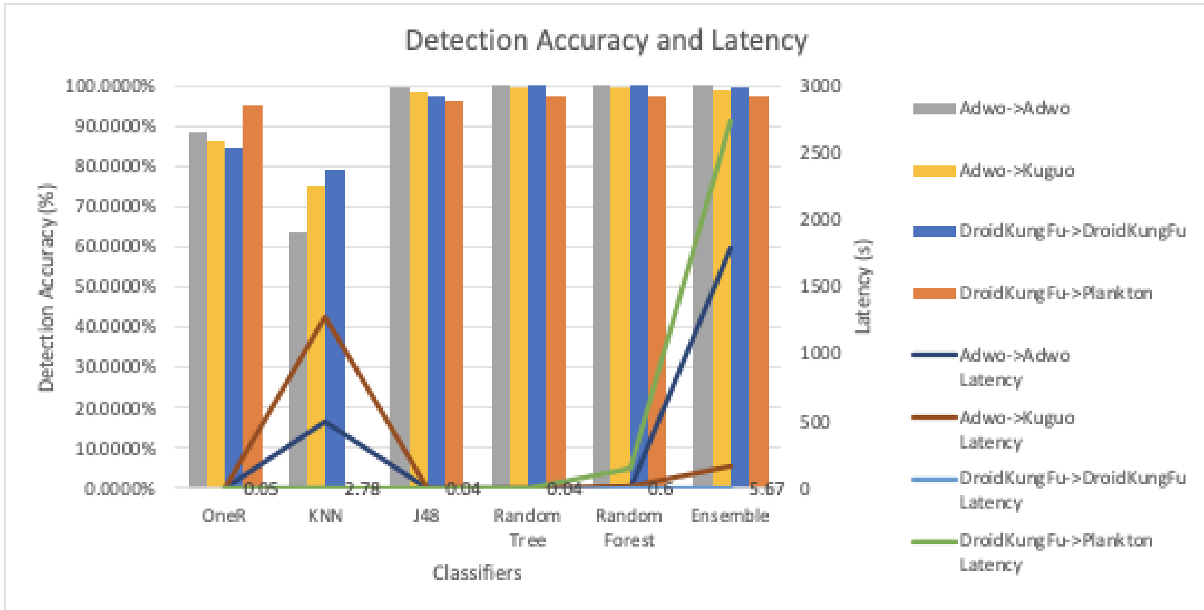

Figure 4.10: Detection accuracy and latency based on each test case (Note: The latency of the classifiers trained with the DroidKungFu dataset and tested with the remainder of the dataset, blue line, is marked with numbers in the figure.)

Random Forest Tree classifier, fits the best for malware detection. As proof, over 96% detection accuracy is achieved from Decision Tree type classifiers (i.e. J48, Random Tree, and Random Forest) in all of the test cases. From the classifiers trained and tested with the datasets in the Adwo group, all the Decision Tree classifiers obtained 99% accuracy. The majority of those Decision Tree classifiers, trained with the Adwo group dataset and tested with the Kuguo group, achieved the same accuracy as well, and the Ensemble classifier has a group of the instances decreased by only 1% in detection accuracy.

The detection accuracy on Random Tree and Random Forest Tree, trained with the DroidKungFu group dataset and classified the DroidKungFu group testing set, is outstanding, and detection accuracy of the Plankton group with the classifiers, trained with the DroidKungFu training set, dropped 2%, but is still remarkable. All the classifiers categorized the instances remarkably well because the benign and malicious versions of Plankton datasets are statically vague, but slightly distinct dynamically, where the role and the effectiveness of machine learning is present. The reason why the Adwo and Kuguo group achieved a better result than DroidKungFu and Plankton group should be

that obfuscated Kuguo applications have far more distinguishable patterns and a larger amount of data as we saw in Figure 4.5 and Figure 4.6. In the case of Ensemble learning, even if KNN achieved lowest detection accuracy, Random Forest seems to leverage the weakness of the KNN classifier, resulting in over 97% for all the test cases.

Despite the high accuracy of Random Forest Tree, latency countered the advantages of the classifier. If the classifiers need to use time sensitive activities, J48 with a little loss of accuracy and less latency, or Random Tree with a higher accuracy and longer latency than J48 can be alternatives to Random Forest Tree. If the accuracy can be flexible to some extent, even the OneR classifier could replace other classifiers, but it could be risky because of the simplicity of the classifier.

Chapter 5
Conclusion

To the best of my knowledge, this is the first approach to testing the application of HPCs in detection of applications with update attack. In this thesis, the target is to research whether the use of HPCs is feasible to detect the update malware, specifically that which is piggybacked. During the study, what the update malware is, how the malware invades into users' realm, what kind of differences exist between regular and piggybacked update malware, and what specific update malware is targeted were explored in depth. The piggybacked applications infected with Adwo, Kuguo, DroidKungFu, and Plankton were chosen as the target due to their capability of easy transformation from benign applications to malicious. This is achieved by embedding the malicious payload based on the characteristic of the piggybacking technique and stealthiness of piggybacked applications. Also, previous work conducted various techniques to detect the wide range of regular malware in applications including Plankton and DroidKungFu (non-piggybacked). A framework was developed for gathering the pool of applications and collecting the HPC data, leaving the prospect for future work. Then, the data collected with HPCs on the applications cannot finalize in detection of malware by itself, leading to the adoption of machine learning. Machine learning was leveraged to determine the classifiers which are the best fit for malware detection. The classifiers that were discovered as the most efficient are Decision Tree and Ensemble learning. The simple machine learning algorithm, such as OneR and KNN, resulted in low detection rate. Specifically, OneR achieved 85%-95%

whereas KNN performed the worst besides the classifiers. J48, Random Tree, and Random Forest Tree accomplished around 94%-99%, and Ensemble learning reached almost 100% detection accuracy, but the latency for Random Forest Tree and the Ensemble learning are significant depending on the size of the testing dataset. To avoid the substantial computation cost, two other Decision Tree classifiers can be alternatively used for malware detection. From the overall results given by the classifiers, hardware performance counters (HPCs) are recommended for discovering stealthy update attacks as a dynamic analysis technique. Also, a future work that can be considered exists: the collection of HPC data takes 15 seconds per cycle, which runs 80 cycles total, resulting in 20 minutes on one application. This time can be reduced by extracting and minimizing the number of the events. Moreover, the choice of lighter computation classifiers can be accumulated on the HPC data and creation of a new security scanning system is possible. This work not only expands the possibility of the usage of HPCs for update malware detection, but also provides the framework for researchers to obtain and test the data with minimized effort for related progressive work.

References

[1] B. Cho, S. Lee, M. Xu, S. Ji, T. Kim, and J. Kim, "Prevention of cross-update privacy leaks on android," 2018.

[2] A. Salem, "Stimulation and detection of android repackaged malware with active learning," *arXiv preprint arXiv:1808.01186*, 2018.

[3] F. Mercaldo, V. Nardone, A. Santone, and C. A. Visaggio, "Download malware? no, thanks." in *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering–FormaliSE*, 2016.

[4] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE symposium on security and privacy.* IEEE, 2012, pp. 95–109.

[5] H. Sayadi, N. Patel, S. M. PD, A. Sasan, S. Rafatirad, and H. Homayoun, "Ensemble learning for effective run-time hardware-based malware detection: A comprehensive analysis and classification," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC).* IEEE, 2018, pp. 1–6.

[6] Y. Zhou and X. Jiang, "An analysis of the anserverbot trojan," *Dept. Comput. Sci., North Carolina State Univ., Tech. Rep*, 2011.

[7] L. Tenenboim-Chekina, O. Barad, A. Shabtai, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici, "Detecting application update attack on mobile devices through network featur," in *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS).* IEEE, 2013, pp. 91–92.

[8] L. Li, D. Li, T. F. Bissyandé, J. Klein, H. Cai, D. Lo, and Y. Le Traon, "On locating malicious code in piggybacked android apps," *Journal of Computer Science and Technology*, vol. 32, no. 6, pp. 1108–1124, 2017.

[9] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C).* IEEE, 2017, pp. 359–361.

[10] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of" piggybacked" mobile applications," in *Proceedings of the third ACM conference on Data and application security and privacy*, 2013, pp. 185–196.

[11] N. Collier. Barcode scanner app on google play infects 10 million users with one update. [Online]. Available: https://blog.malwarebytes.com/android/2021/02/barcode-scanner-app-on-google-play-infects-10-million-users-with-one-update/

[12] K. Basu, P. Krishnamurthy, F. Khorrami, and R. Karri, "A theoretical study of hardware performance counters-based malware detection," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 512–525, 2019.

[13] ARM. About the performance monitoring unit (cortex-a7 mpcore technical reference manual). [Online]. Available: https://developer.arm.com/documentation/ddi0464/d/Performance-Monitoring-Unit/About-the-Performance-Monitoring-Unit

[14] ——. About the performance monitoring unit (cortex-a9 pmu). [Online]. Available: https://developer.arm.com/documentation/ddi0388/i/performance-monitoring-unit/about-the-performance-monitoring-unit

[15] A. H. Lashkari, A. F. A. Kadir, H. Gonzalez, K. F. Mbah, and A. A. Ghorbani, "Towards a network-based framework for android malware detection and characterization," in *2017 15th Annual conference on privacy, security and trust (PST)*. IEEE, 2017, pp. 233–23 309.

[16] A. Salem and A. Pretschner, "Poking the bear: Lessons learned from probing three android malware datasets," in *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis*, 2018, pp. 19–24.

[17] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

[18] N. Sharma, A. Chakrabarti, and V. E. Balas, "Data management, analytics and innovation," *Proceedings of ICDMAI 2019*, vol. 1, 2019.

[19] E. Ahmet and S. H. S. Hussin, "Malware visualization techniques," *International Journal of Applied Mathematics Electronics and Computers*, vol. 8, no. 1, pp. 7–20, 2020.

[20] Y. Yang, X. Du, Z. Yang, and X. Liu, "Android malware detection based on structural features of the function call graph," *Electronics*, vol. 10, no. 2, p. 186, 2021.

[21] A. Pektaş and T. Acarman, "Deep learning for effective android malware detection using api call graph embeddings," *Soft Computing*, vol. 24, no. 2, pp. 1027–1043, 2020.

[22] K. V. Nguyen, "Application update using multiple network connections," Jul. 28 2020, US Patent 10,725,768.

[23] W. Yufeng, R. Linghe, J. Li, and M. Lin, "Application update method and apparatus," Dec. 22 2020, US Patent 10,871,953.

[24] R. S. Vanblon, G. Zaitsev, and J. Zhang, "Application update control," Mar. 3 2020, US Patent 10,579,360.

[25] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici, “Mobile malware detection through analysis of deviations in application network behavior,” *Computers & Security*, vol. 43, pp. 1–18, 2014.

[26] I.-K. Park and J. Kwak, “Implementation of permission management method for before and after applications the update in android-based iot platform environment,” 2017.

[27] A. Salem, F. F. Paulus, and A. Pretschner, “Repackman: A tool for automatic repackaging of android apps,” in *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis*, 2018, pp. 25–28.

[28] Android. Simpleperf introduction. [Online]. Available: https://android.googlesource.com/platform/prebuilts/simpleperf/+/782cdf2ea6e33f2414b53884742d59fe11f01ebe/README.md

[29] B. Gulmezoglu, A. Zankl, T. Eisenbarth, and B. Sunar, “Perfweb: How to violate web privacy with hardware performance events,” in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 80–97.

[30] J. Wang. Android malware dataset. [Online]. Available: https://github.com/traceflight/Android-Malware-Datasets

[31] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, “Androzoo: Collecting millions of android apps for the research community,” in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 468–471.

[32] S. R. Group. Original apps mapped with piggybacked apps. [Online]. Available: https://github.com/serval-snt-uni-lu/Piggybacking/blob/master/ground-truth/all-pairs.csv

ProQuest Number: 28419158

INFORMATION TO ALL USERS
The quality and completeness of this reproduction is dependent on the quality
and completeness of the copy made available to ProQuest.

Distributed by ProQuest LLC (2021).
Copyright of the Dissertation is held by the Author unless otherwise noted.

This work may be used in accordance with the terms of the Creative Commons license
or other rights statement, as indicated in the copyright statement or in the metadata
associated with this work. Unless otherwise specified in the copyright statement
or the metadata, all rights are reserved by the copyright holder.

This work is protected against unauthorized copying under Title 17,
United States Code and other applicable copyright laws.

Microform Edition where available © ProQuest LLC. No reproduction or digitization
of the Microform Edition is authorized without permission of ProQuest LLC.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346 USA