University of South Alabama

JagWorks@USA

Theses and Dissertations

Graduate School

12-2021

Repackaged Android Application Classification Through Static Global Image Feature Analysis

Robert Cox

University of South Alabama, rcc1622@jagmail.southalabama.edu

Follow this and additional works at: https://jagworks.southalabama.edu/theses_diss

Part of the Computer Sciences Commons

Recommended Citation

Cox, Robert, "Repackaged Android Application Classification Through Static Global Image Feature Analysis" (2021). *Theses and Dissertations*. 3.
https://jagworks.southalabama.edu/theses_diss/3

This Thesis is brought to you for free and open access by the Graduate School at JagWorks@USA. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of JagWorks@USA. For more information, please contact mduffy@southalabama.edu.

REPACKAGED ANDROID APPLICATION CLASSIFICATION THROUGH STATIC GLOBAL IMAGE FEATURE ANALYSIS

A Thesis

Submitted to the Graduate Faculty of the
University of South Alabama
in partial fulfillment of the
requirements for the degree of

Master of Science

in

Computer Science

by
Robert Cox
B. S., University of South Alabama, 2019
December 2021

ACKNOWLEDGMENTS

I would like to thank Dr. Jeffrey McDonald, Ms. Angela Clark, and Dr. Todd Andel for affording me the opportunity to participate in the Scholarship for Service. I would like to thank Dr. Ryan Benton for his mentorship and advising. Without their guidance, dedication, and the opportunities they afforded me, graduate school would not have been a possibility. Last but certainly not least, I want to thank Dr. Straub for being a part of my committee and for the attitude, enthusiasm, and general awesomeness that he always brings to the classroom.

TABLE OF CONTENTS

LIST OF TABLES v

LIST OF FIGURES vi

LIST OF ABBREVIATIONS vii

ABSTRACT viii

CHAPTER I INTRODUCTION 1

CHAPTER II BACKGROUND AND RELATED WORKS 4

2.1 The Problem 4
2.2 Machine Learning 5
2.3 Visualization 6
2.4 Malware Classification 7
2.5 Detection of Repackaged Applications 13

CHAPTER III PROPOSED RESEARCH AND METHODOLOGY 20

3.1 Basic Approach 21
3.2 Tools 22

3.2.1 SKLearn 22
3.2.2 PIL 22
3.2.3 OpenCV 24

3.3 Features 24

3.3.1 Haralick Texture 24
3.3.2 Hu Moments 25
3.3.3 RGB vs HSV Color Histogram 26

3.4 Description of Experiments 27

CHAPTER IV RESULTS 30

4.1 Replication Experiments 30
4.2 Repackaging Results 34

CHAPTER V CONCLUSION 44

5.1 Importance 44
5.2 Significance of Results 46

REFERENCES 48

BIOGRAPHICAL SKETCH 53

LIST OF TABLES

Table Page

1. Machine Learning Algorithms Used ... 23

2. Size of Each Data Set ... 28

3. Training and Testing Combinations for Repackaging Detection ... 28

LIST OF FIGURES

Figure Page

1. Hu Moments Definition 25

2. Results of First Malware Detection Experiment.. 32

3. Results of Second Malware Identification Experiment 33

4. Results of First Repackaged Application Detection Experiment................................. 34

5. Results of Second Repackaged Application Detection Experiment 35

6. Results of Third Repackaged Application Detection Experiment 36

7. Results of Fourth Repackaged Application Detection Experiment 37

8. Results of Fifth Repackaged Application Detection Experiment 38

9. Results of Sixth Repackaged Application Detection Experiment................................ 39

10. Results of Seventh Repackaged Application Detection Experiment 40

11. Results of Eighth Repackaged Application Detection Experiment 41

12. Results of Ninth Repackaged Application Detection Experiment.............................. 42

13. Results of Tenth Repackaged Application Detection Experiment.............................. 43

14. Past and Projected Growth of Smartphone Users (In Billions) 45

15. Summary of Repackaging Detection Results 47

LIST OF ABBREVIATIONS

Hard Disk Drive .... HDD

Operating System .... OS

Android Package Kit .... APK

Application Programming Interface .... API

Scale-Invariant Feature Transform .... SIFT

Speeded Up Robust Features .... SURF

Oriented Fast and Rotated Brief .... ORB

Python Imaging Library .... PIL

Dalvik Executable .... DEX

Bitmap Image File .... BMP

Features From Accelerated Segment Test .... FAST

ABSTRACT

Cox, Robert, M.S., University of South Alabama, December 2021. Repackaged Android Application Classification Through Static Global Feature Analysis. Chair of Committee: Ryan, Benton, Ph.D.

Over the last 10 to 12 years, Android device usage has shown a steady increase year over year. This leads to the same growth, if not more, in Android application availability and usage [1]. Based on the drastically increased usage and the ease of reverse engineering, malicious repackaged applications have become a severe problem within the Android Marketplace. One potentially fruitful approach is the use of a combination of visualization, pattern recognition, and machine learning methods to classify applications as either malware or safe. The first part of this research focuses on an image-based malware classification system based on static analysis of visualizations by means of feeding global features extracted from images to machine learning and prediction algorithms. Results that incorporate a subset of the proposed features indicate we can achieve 85% malware classification accuracy. The second part of this research focuses on using the same technique to identify repackaged applications rather than specifically malicious ones. A series of experiments with different combinations of training and testing images showed an accuracy of up to 99.7% in identifying an application that has been repackaged.

CHAPTER I

INTRODUCTION

Can image classification and prediction be applied to static images of applications to detect embedded malicious code and/or pirated applications with good accuracy and computational efficiency? Can certain extracted image features be used to accurately predict malware by machine learning and classification? These are questions that if answered could be a big step in malicious Android App detection. With the Android operating system and applications at their highest usage yet and no sign of the growth slowing down for the next several years, maliciousness within apps is becoming a bigger problem than ever [1]. The use of sensitive information through Android apps is higher than ever as well, creating even more opportunities for an attacker to benefit from a single attack. This leads to a variety of concerns when considering the potential methods and magnitude of an attack [1].

Machine learning has also drastically evolved over the years and is currently one of today's most rapidly growing fields [2]. At the rate data has been produced in recent years, there have been many different opportunities generated for researchers to fine tune the fundamental laws governing learning and to develop more solid algorithms. Such algorithms can even be considered essential during this time of seemingly exponential data production. When we think of our modern technologies such as self-driving

automobiles, computer vision, speech recognition, modeling, and many newer abilities within the medical field, we must think of machine learning [2]. With all of the proven uses of machine learning and the fact that we have barely scratched the surface of the possible uses, it is believed that various protection measures such as identifying repackaged and/or malicious software can and will be developed to better protect from attacks on user data and the pirating and redistribution of proprietary software applications.

Machine learning methods can be used in both static and dynamic analysis. Static analysis does not require a runtime environment as it does not focus on functional testing. It involves analyzing the code, either source, byte, or binary to look for indicators of vulnerabilities or malicious insertions. This means using the structure of the application without executing it to identify suspicious code. Dynamic analysis does require a runtime environment because it does focus on functional testing. It is a method of executing the application under various conditions or sets of conditions and analyzing the behavior to look for indicators of vulnerabilities or malicious insertions. The main disadvantage to the dynamic approach is, of course, executing potentially or known malicious applications. Both methods have strengths and weaknesses that can be used in combination to identify these vulnerabilities or malicious insertions more accurately.

The goal of this research is to implement and evaluate a system to detect malicious and repackaged Android applications through the use of static analysis and supervised machine learning. To achieve this, the Android application files are converted from their inherent binary code into an image representation of each file. Next, image processing techniques are used to extract features from each image; these features are

then used to train machine learning algorithms to use those features to classify an application as either repackaged or original. If repackaging can be accurately detected, that can automatically save users from malicious Android applications because the majority of malicious applications are repackaged [1]. The significance of this research is potentially huge, as it could have a substantially positive impact on a problem that has been consistently increasing while making it much more difficult for attackers to successfully compromise the operating system that composes 85% of the world's mobile market [1].

This chapter introduced the growth of Android operating system usage and the associated growth attacks. This chapter also introduced the concept of machine learning, static and dynamic analysis, and the overall goal of this research. In Chapter II, related concepts and background information on related research are discussed.

CHAPTER II

BACKGROUND AND RELATED WORKS

2.1 The Problem

Android's operating system was released in 2008, and by mid-2009, it made up approximately 3% of the smartphone market. By mid-2010, it composed almost 10% of the smartphone market, by 2011, that number was almost 33% of the smartphone market, and today, the Android's operating system makes up approximately 85% of the world's mobile market [3]. This operating system is used on a variety of devices including smartphones and tablets and the number of available Android applications has shown incredible growth. In 2019, there were nearly 20,000,000 Android malware samples, which was up 31% from the previous year. There is currently no foreseeable reduction in the growth of Android usage through at least 2023 [1]. Based on the growth and popularity, there has been an increase in the number of applications that use and store sensitive data. The increase in the volume of sensitive data results in an increase in the number of users who become dependent on these devices and applications, which increased the appeal to an attacker or malicious user. This results in a much higher need for security measures. It is quite clear that neither the personal nor the commercial use of mobile devices is going anywhere. This is where techniques such as the ones reviewed

and evaluated in this research project come into effect and new techniques with higher accuracy and computational efficiency are needed.

Attackers can be very creative, and even with the current processes for analyzing Android applications, there are many issues that remain including cumulative attacks (an attack where the attacker repeatedly attempts to attack the application, and though they fail many times, they learn something new about the application with each attempt and can eventually use the accumulated knowledge to achieve a successful attack). An attacker does not need to implement a malicious attack on the user to be successful. Attacks can include removing ads from the application or redirecting legitimate funds from an ad into a personal bank account or bypassing required authentications that exist in purchased or free applications that contain in-app purchases to allow users to receive the paid services for free. Although open-source and commercial repackaged application detection systems exist, they have been proven ineffective and better methods are needed [1].

2.2 Machine Learning

Machine learning carries an incredible significance to solving today's problems and can be thought about in various ways. Jordan and Mitchell [2] define machine learning as a discipline focused on constructing computer systems that automatically improve through experience fundamental statistical-computational-information-theoretic laws that govern all learning systems, including computers, humans, and organizations [2]. Over the past twenty years, machine learning has progressed from a concept being explored in a laboratory to widespread practical and commercial use [2]. The most

widely used machine learning algorithms are known as supervised learning methods [2]. This means that they exemplify function approximation to achieve a result. Function approximation is where the task is embodied within a function and the learning efforts are to improve the functions accuracy with training data, also called known input-output pairs. Typically, the more training data that is available, the more accurate the learning algorithm will be when making a prediction [2]. Although many major breakthroughs have already occurred, the machine learning field is relatively young and still expanding, including the trend of expanding the environment in which many of the algorithms operate. One example of environment expansion is the number of processors used for a learning algorithm and focusing on parallel implementations [2].

2.3 Visualization

Kartel et al. [4] perform an analysis of visualization techniques used for malware detection. Visualization driven techniques have illustrated a strong potential to increase the efficiency of the malware analysis and identification process. While there have been many research efforts and developments in the visual analysis of network traffic and security events, that is not the case for malware visual analysis systems. Malware analysis can be organized into three main categories: Individual malware analysis to gain a new understanding of individual behavior, malware behavioral comparisons based on either a system of sequence calls or based on generated images of malicious software, and malware summarization. Summarization systems summarize the behavior of various malware samples in effort of making discoveries pertaining to the overall behavior [4]. Kartel et al. [4] identified malware analysis task, source data, input data format,

visualization techniques, and interaction as criteria for comparing the different visual analysis systems.

2.4 Malware Classification

Venkatraman and Alazab [5] designed a malware classification system using visualization of similarity matrices. Visualization of two-dimensional similarity matrices is a technique often used to measure similarity between two objects. Traditionally, it is commonly used to visually represent documents and measure the similarity for reasons such as detection of plagiarism. In their paper, Venkatraman and Alazab [5] apply this technique to a similarity matrix between different commonly employed malwares. They also use this technique to compare a malware dataset with a benign dataset to demonstrate their differences [5].

Their model consists of 3 key steps. First, they use reverse engineering tools to disassemble an executable program, each decomposition representing a vector of functions *F*. *F'* is the variant malware of the original executable, also represented as a vector of functions. Second, the similarity between program *F* and malware *F'* is computed through similarity and difference mining, creating a similarity matrix for each. Lastly, a comparison is made between the similarity matrix values and the threshold values to determine if the program is malicious. Comparing with other samples can also help with identifying groups or families of malware. The experiment ran on three different processors, and the similarity was evaluated using a very large real-life malware dataset. The datasets were obtained through public databases and the similarity measures were able to accurately identify all malware variants [5].

Chen et al. [6] apply image classification technology to Android malware. In their work, they visualize Android programs and build a machine learning model that used image processing technology to classify the images. After development, Android files are packaged into APK files, which is basically a ZIP package. The first step of the visualization process is to decompress the APK files to view the contents of them. Once decompressed, the DEX files which encapsulate the source code of the Android program can be processed and converted into an image representation for features to be extracted to train the machine learning model [6].

There is one main feature used for this classification experiment, the Gist feature. The Gist algorithm is commonly used in image recognition systems. The Gist feature simulates the image extracted by the human visual system. Gist defaults to four scales and eight directions of Gabor filter banks, extracting the global features of the image [6]. Instead of processing the entire image at once, Gist uses a sort of divide and conquer approach. First it blocks the image, then extracts the features from each block using a set of filters, and lastly combines them back together [6].

For their classification model, Chen et al. [6] used XGBoost algorithm, which is an implementation of gradient boosted decision trees designed for speed and performance. For the performance comparison and validation, XGBoost was evaluated and compared to KNN (K-Nearest Neighbors) and GBDT (Gradient Boosting Decision Tree) with 10-fold cross validation according to accuracy, precision, recall, and F1score. XGBoost achieved a 99.1% accuracy, which was the highest of the four. The next best accuracy rate was KNN, reaching 98.1% [6].

Wang et al. [7] propose a static analysis-based solution to detecting maliciousness within Android applications, however their solution takes into consideration the false negatives that easily arise by using only one type of feature set. Their solution, called DroidEnsemble, uses both structural features and string features for a static analysis. String features include things like permissions, hardware, filters, and API interactions. Structural features include things such as the flow of data and controls (function calls are very useful). Their solution uses three of the same machine learning algorithms that are in this research: K-Nearest Neighbor, Support Vector Machine, and Random Forest [7].

Wang et al. [7] solution consists of four main steps. The first step is collecting applications from randomly selected markets and malapps. Next, six string features are extracted from each of the applications, as well as a function call graph from each one. Third, the machine learning models are constructed using the features that were extracted in step two, all of which are supervised. Once the machine learning models are constructed, they validate the accuracy of the models [7].

The string features that they use are explained in detail. The first feature is requested permissions. A requested permission is something that an application must obtain for certain operations that it is performing, however if the permission is determined to be unnecessary for this operation, that could easily be deemed as a potentially malicious action. The second is hardware features. These are indicators of things that the application is using or demanding, such as the location. The third string feature is filtered intents. Inter-component communication is handled by intents, and filters allow the wanted communications and reject the unwanted ones. The fourth is restricted API

calls. Permissions protect these restricted API calls, as they are considered critical, and the calls are identified by permission maps. The fifth feature is the used permissions. These are the permissions that were used/passed and are also identified by the permissions map. The last string feature is code patterns. Portions of the malicious code that attackers have inserted can be concealed within existing libraries. An application's library, shell command, and external file usage patterns can indicate malicious behavior [7].

Wang et al. [7] authors take these six string features and combine them with structural features. The structural features consist of derivations from function call graph analysis and comparison. These graphs are extracted from the applications once they have been disassembled. The function nodes are categorized and labeled with 15-bit encodings, and hashes are calculated between the caller of each function and the function itself. Their experiments are conducted on real-world applications. Using only the string features, the authors were able to achieve a 95.9% accuracy. Using only the structural features, they were able to achieve a 90.68% accuracy. Once combined, they were able to achieve up to a 98.4% accuracy. The data for these experiments consisted of 1386 benign applications and 1296 malicious applications [7].

In part of this research, there is an attempt to recreate results obtained by Unver and Bakour [8] for malware classification without the local features and using the same global feature set and methodology on our data set before running experiments with classifying repackaged Android applications. This research also includes an attempt to achieve better results than that obtained by Minor [9] for malware classification, who researched the impact of spatial layout methods when using visualization techniques for

color and grey scale. Unver and Bakour [8] solution includes the visualization step but converts the Android files to only grey scale images. Unver and Bakour [8] also use a different method of mapping the bytes to the images than Minor [9] did in his previous research. Another way their solution differs from Minor's [9] is that it includes two additional global features and four local features. Minor [9] used Haralick texture alone, Unver and Bakour [8] used Haralick Texture, Hu Moments, and Color Histogram as their global feature set that gets extracted from the images, which is what this research attempts on malware classification. The purpose in experimenting less the local features is to determine if the same results can be generated with less computational effort. Their local features (which were omitted in the current research) are SIFT, SURF, KAZE, and ORB [8].

SIFT is an algorithm that determines key points in an image, also known as the minima and maxima. SURF is an algorithm that is an alternative or variation of SIFT. SURF uses Hessian matrix, while SIFT uses Gaussian Log. SURF tends to be faster than SIFT, however SURF is less detailed. KAZE is a variant of SURF, as it also uses Hessian matrix, but does so with a normalized determinant. ORB detects key points with the FAST algorithm and then uses Harris corner algorithm on the key points that it detected. One of the main limitations that Unver and Bakour encountered was the ability of code obfuscation to have a negative impact on the accuracy, however they speculate that this obstacle can be removed by including semantic features from the code. That speculation is left to future work. Overall, they were able to obtain a 98% accuracy rate in classifying the malicious Android applications [8].

Jung et al. [10] propose a solution that utilizes various characteristics of Android application programming interface communication. In order for an Android application to accomplish any objective, a series of communications between select APIs is required. Examples of objectives that require these communications are memory, process, and IO management, and graphic processing. Jung et al. [10] exploit this required communication and categorize APIs accordingly. They create two categories: one for APIs considered to be benign and one for APIs considered to be malicious. The benign and malicious API lists are formed from documenting the APIs that are most commonly called within vetted applications and the APIs that are most commonly called within known malicious applications. These lists are also ranked, where the API called the most in each list is ranked highest and the API called the least in each list is ranked lowest. An example of a lowest ranked API is one that is called only once in the entirety of each Android application set [10].

Once these two ranked lists are constructed, the begin processing this data. They do two things with this data: Keep a copy of the ranked lists as is and then take these two lists and remove any APIs that they have in common creating a new set of lists that are smaller and more tailored to the type of application that they were derived from. Using both sets of lists (the pre-processed ones and the processed ones), they determine which APIs are called upon and use the ranking (1, 2, 3, 4, etc.) from each of the two lists to calculate a score. They obtain this score by calculating the inverse sum from the ranking of each of the lists, and the application takes the label (benign or malicious) of the higher score. So, if three APIs are identified, and each API is ranked 2, 4, 8, and 2, 3, 12 in the benign and malicious categories respectfully, then the score for the application being

benign is $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} = 0.875$ and the score for the application being malicious is $\frac{1}{2} + \frac{1}{3} + \frac{1}{12} = 0.9166$. This is a case where the higher score comes from the malicious list, thus categorizing this application as malicious [10].

With the first option, using the ranked lists as-is (before removing the APIs that the two lists have in common), the top 600 APIs from each list are selected, 398 of which are common to both lists but not removed. This method yields them a detection accuracy of 87.85%. A second experiment is executed using the top 700 APIs, 463 of which are common to both lists but are not removed and yield an 87.35% detection accuracy. With option two, the top 700 APIs are used, and the 463 that are common to both lists are discarded leaving two lists of size 237 that have nothing in common. This experiment yields them an 89.93 detection accuracy, a little higher than option one. It is mentioned that option two, due to the lack of ambiguity in the training data, would be better for machine learning algorithms in future work [10].

2.5 Detection of Repackaged Applications

Most often, repackaged applications are variants of widespread applications that have been infected with malware to infiltrate users' mobile devices. Attackers will download a popular application, reverse engineer it, and add their own malicious code. Once that is accomplished, they will release their malicious version of the application. If repackaged Android application detection can be effectively implemented, then by default the users can be saved from a large portion of malicious Android applications since most

infected Android applications are repackaged and Android is the mobile OS that is targeted the most [11].

Currently, there are three recognized categories of repackaged applications, each being decided by the purpose of the attack [12]. The first category is malware. To be defined as malware, malicious intent is required, and malicious code is typically inserted within legitimate code to accomplish the malicious task. Attackers will most often download the original application, reverse engineer it, insert their code, then repackage and redistribute it. The second category is fake advertisement. Song et al. [12] mention several examples of fake advertisement. The first is someone playing a video game and altering the code to remove the ads, the second is creating new ads and re-routing the money generated from those ads for their own benefit. The last category is in-app purchase crack. This category is when an attacker is able to defeat the authentication part of an application that validates it was paid for, thus receiving the application or services for free [12].

Nguyen et al. [1] proposed a novel repackaged application detection system that uses the perceptual hashes and dynamic user interface behavior of Android applications that are known to be legitimate. Malware that has been repackaged within an Android application can operate in a dormant manner allowing it to be stealthy and go un-noticed by the user. Most of the repackage detection research has focused on static analysis without having to run the code, however their approach is dynamic and involves executing the malicious application. The dynamic approach helps to rid some of the obstacles of the static approach such as obfuscation and encryption. One thing unique about their approach is that during the execution they use a runtime emulation to capture

images of the application. Perceptual hashes can be used to measure the similarity of two images. In their research, they used three different perceptual hashing algorithms: Average Hash, Perception Hash, and Difference Hash. Average Hash proved to be the most accurate in detecting the repackaged applications carrying an 88.16% accuracy rate [1].

Chen et al. [13] propose a solution for detecting repackaged apps that is different than the two primary ways that currently exist. First, they explain how one of these ways of detecting repackaged apps relies on self-check sum. In a self-check sum part of the code calculates a precise value (typically using a hash) that represents the code exactly when it was originally completed and then checks that value each time the application is executed to verify that it matches the original value every time. If the values don't match at any given runtime, then the application has been altered). The second method is to compare and look for differences in the respective APK files, which is typically done by the market that the APK file is being distributed from. Since some markets do not actually check for repackaging, their solution involves users working together in a web-based approach that allows various levels of user collaboration [13].

Their system is composed of five modules, which flow like this: Clients have a connection established with the server, and there is an upload module that allows for the feature data from all of the applications installed in the user's phone to be uploaded. The request module then requests a detection to be initiated. The detect module then proceeds with the detection process, and the assess module runs an assessment of the risks right away and feeds the results to the last module, called the display module. The collective

user base is responsible for consistently uploading and updating to the online system, and according to Chen et al. [13], this allows for advantages not typically seen.

The first advantage is that is does not require extra human involvement to detect new repackaged applications in a timely manner. The strength of the system is heightened due to the constant collaboration of many users and the frequent updating of the data used for the detection. The upload module will not upload the data from system apps since they are tailored to the mobile device by the manufacturer. Their experiments used 1684 app feature data from 50 mobile phones, with an average of 33 applications per phone. Their system detected eight repackaged applications on six different phones, and it also detected that one application had been expanded in markets. Their detection method is executed post app installation and was able to conquer the shortcomings of markets and in-app self-checking [13].

Sun et al. [14] also stress how much the Android user base continues to grow and how unreliable a market or other third party can be when it comes to detecting or even actually attempting to detect repackaged applications, along with how common a false negative is in the typical market approach. They propose an active approach for the detecting of repackaged Android applications. Their approach utilizes embedded watermarking within the section of code that is responsible for the detection efforts, specifically within conditional branches. This method consists of three main modules: one module for identifying branches that are candidates, one module for creating the watermarking and embedding it within the code, and a third for encryption and application packaging [14].

For the first module, all of the conditional branches are scanned for and recorded. Once recorded, then changes are made to those branches and as a result the code is changed. Next, candidates are selected according to the presence of a constant. The constants are simultaneously recorded in a table which is to be used for watermarking encryption. For the second module, they generate watermarking by encrypting and compiling both the original code and the detection code. The signature of the Android application is the key to determining the packaging. Their steps to the watermarking are quoted as follows: "implementing the libSDC library, scanning application source code to search for candidate branches, compiling a separate file extracted by C to form a DEX file, decrypt that file and dynamically load and run it." For the last module, they use the keys (the constants that they inserted in the conditional branches) from the branch table to encrypt the different watermarks. They ran experiments on ten different Android applications that were retrieved from Github. Their experimental results show effectiveness and feasibility, but the accuracy needs improving, and their next step is to work on developing and embedding more watermarking types to accomplish the desired increase in accuracy [14].

Another major concern with existing Android application anti-repackaging techniques is that they cannot meet the requirements in terms of performance, nor can they handle a lower granularity of protection such as multiple repackaging techniques working together to repackage an application. Song et al. [12] introduced AppIS, a repackaging detection system that handles some of these concerns. Rather than the standard and rather inefficient approach that provides simple protection and typically considers only one

aspect, their design provides time diversity and multiple levels that interlock with one another. Their guarding mechanisms are also randomly constructed using a dispatcher and controller. Their architecture makes use of the traditional guarding net and has three main phases. The first is pre-processing, the second is static guarding net construction, and the last is dynamic guarding net construction [12].

Their design specifically targets the Java programming language since it is the primary language used for Android programming, and they also use Java to implement their design. Song et al. [12] say that guards should protect each other and have designed a structure that that have several guards composed of the trigger module in the java layer, the main function module in the native layer, and the key information module. Because the java layer has been proven easy to penetrate, their main function module is coded in C++ and the triggering is coded in JNI. They create three guards: The first is called J_Guard. JGuard is used to protect Java code that was written by the user. The second is called N_Guard. N_Guard is used to protect native functions of other guards. The last one is called D_Guard. D_Guard is used to protect and reinforce a DEX file [12].

AppIS uses multiple dimensions and a guarding net that interlocks to separate their design from using a series of single guards, which has proven to be feasible. This means that in the event of a compromised guard, there are other guards that can be deployed to come to the rescue and implement measures in response. Another unique feature of their design from others is that it is able to better protect against a cumulative attack. They can defend against cumulative attacks by implementing a dynamic and randomized scheduling system. For each attack, the scheduler calls upon a different set of guards, thus creating a different guarding net so the attacker can’t repeatedly attack the

same guarding net and learn about it. Theory and limited experiments have shown their model to be feasible and valid, but more experiments are required to truly evaluate the effectiveness [12].

The visualization and image feature extraction approach has been attempted on classifying Android malware, but with limitations. One of the main limitations highlighted by Unver and Bakour [8] is that the performance of this approach may be hindered by code obfuscation. A second limitation that they highlight is how this approach is not effective against injection attacks. The potential remedy that is attempted in this research is detecting the repackaging of Android applications rather than only the maliciousness. Due to the fact that an attacker has to repackage the application in order to reinsert it into the market regardless of what has been done within the application, code obfuscation and injection could make much less of a difference in the detection accuracy.

This chapter discussed the overall concern with cybersecurity was discussed. This chapter also discussed related work and used concepts including machine learning, visualization, malware detection, and repackaged application detection. Chapter III discusses this research methodology and the tools and features used in the experiments.

CHAPTER III

RESEARCH AND METHODOLOGY

In this chapter, the data used is discussed, along with the approach to running the experiments, and the tools and features used to accomplish and evaluate the experiments. The machine learning algorithms that are used are also discussed and described. For a preliminary experiment, PIL is used to create new images from both the malicious and benign DEX file set. Next, the Haralick Texture, Hu Moments, and the Color Histogram are extracted from these images. This extracted data is used to train the machine learning algorithms to classify the images in their respective categories, and ten-fold cross validation is used to evaluate the effectiveness of the training. The machine learning algorithms that are used, the data set sizes, and the combinations used in the experiments are discussed.

The same methodology for the targeted experiments is also used for the preliminary experiments. There were three objectives to accomplish, the first two being measured by the preliminary experiment. The first objective was to achieve better results with malware classification than in previous research conducted by Minor, but with more global features. The second objective is to recreate results achieved by Unver and Bakour [8], but without their local features described in the previous section. The final objective requires conducting a series of experiments are conducted to measure the

effectiveness of this method to detect whether or not an Android application has been repackaged, regardless of its threat category.

3.1 Basic Approach

The choice of programming language is Python due to the variety of machine learning and data manipulation modules that it offers and how user friendly it is. Sets of both benign and malicious applications were provided by Minor's research [9]. Samples of both malicious and benign pre-created Android application images were also provided by Minor. Each of the global features from each image in each category is input into seven different machine learning classification algorithms in Python's Sci-Kit Learn (SKLearn) module in effort of training the algorithm to accurately classify whether an image is infected or uninfected. Once the algorithms are trained, ten-fold cross validation is performed. Cross-validation is a technique to evaluate predictive models by partitioning the original sample into a training set to train the model, and a test set to evaluate it. Ten-fold means that the original samples are partitioned into ten different subsample sets of equal size, one of which is made to be a testing set. This process is repeated ten times with each of the subsamples used exactly once. Once cross validation is complete, a performance histogram containing the accuracy of all seven algorithms is displayed.

3.2 Tools

In this section, the tools used in this research are covered. The tools chosen are either implemented in Python or have Python wrappings. These include SKLearn, PIL, and OpenCV. Each tool will be discussed in detail in its own section.

3.2.1 SKLearn

SKLearn is a machine learning module in the Python programming language. It was built on NumPy, SciPy, and matplotlib, and provides simple and efficient tools for predictive analysis methods such as classification, regression, clustering, etc. It provides many methods for data manipulation and preprocessing such as dimension reduction, model selection, and preprocessing measures such as the feature extraction methods used in this research project. SKLearn is open source and freely accessible. This research focuses particularly on using many of the classification algorithms provided by SKLearn on the global feature data extracted from images. Each algorithm differs in the approach that it takes in classification, and the following table provides the name and a brief description of each algorithm used [15].

3.2.2 PIL

The PIL enables the Python interpreter to have image processing capabilities. This library provides extensive file format support, an efficient internal representation, and image processing abilities that are quite powerful. PIL provides the methods that are used to take the binary data that are read from the DEX files that are extracted from the APK executables and map them to images in the RGB (Red, Green, Blue) color space [16].

Table 1. Machine Learning Algorithms Used

Logistic Regression	Predict the probability of a categorical dependent variable. the dependent variable is a binary variable that contains data coded as 1 (yes, success, etc.) or 0 (no, failure, etc.) [17].
Linear Discriminant Analysis	Estimates the probability that a new set of inputs belongs to each class. The class that gets the highest probability is the output class and a prediction is made [15].
K-Neighbor Classifier	It is a lazy learning algorithm since it doesn't have a specialized training phase. Rather, it uses all of the data for training while classifying a new data point or instance [17].
Decision Tree Classifier	The main advantage of the decision tree classifier is its ability to use different feature subsets and decision rules at different stages of classification. ... A decision tree normally starts from a root node, and proceeds to split the source set into subsets, based on a feature value, to generate subtrees [15].
Random Forest Classifier	It can be used both for classification and regression. It is also the most flexible and easy to use algorithm. Random forest creates decision trees on randomly selected data samples, gets prediction from each tree and selects the best solution by means of voting [15].
Gaussian Naïve Bayes Classifier	A variant of Naive Bayes that follows Gaussian normal distribution and supports continuous data. Naive Bayes are a group of supervised classification algorithms based on the Bayes theorem. It is a simple classification technique and has high functionality [17].
Support Vector Machine	A supervised classification algorithm that is used to classify binary and categorical response data. A model is fit on a set of training data, then the model is used to predict the categories of new observations [17].

3.2.3 OpenCV

OpenCV is an open-source machine learning and vision library. This library carries a variety of classic and modern algorithms for many uses such as facial recognition, object identification, movement tracking and classification, image similarity detection, model extraction, and many other uses. It has C++, Python, Java and MATLAB interfaces and supports Windows, Linux, Android, and Mac OS. OpenCV is the library that provides the methods for image reading, feature extraction, and some of the manipulation required for the parameters of the classification algorithms [18].

3.3 Features

3.3.1 Haralick Texture

Haralick et al. [19] describe the texture and tone of an image. Tone and texture exist in all images, but one can dominate the other depending upon the image. In cases where a small-area patch of an image has little variation of discrete gray tone features, tone is the primary feature. In cases where a small-area patch has a wide variation of features of discrete gray tone, texture is the primary feature. The size of the small area patch makes all the difference between the two. The spatial pattern of the resolution cells is important in determining whether the image carries a fine-grain or course texture. When there is no spatial pattern and the gray-tone variation between features is wide, there is fine texture. When the spatial pattern is more definite and involves more and more resolution cells, the texture is more course [19]. Haralick et al. [19] developed an algorithm to quantify the measurement of texture by using what he called gray tone-

spatial-dependence matrices. One of the main aspects of his framework are the four closely related measures that derive all of the texture features, which is a total set of 14 measures of textural features. The algorithm expects the image to be gray scale [19].

3.3.2 Hu Moments

Peleshko et al. [20] review invariants for intellectual tasks of image processing. Invariants are image properties that remain unchanged after certain transformations are performed on the image. Geometric moments were first introduced by Hu and are based on algebraic invariants. Central moments are calculated from the center of an image, and there are seven Hu Moments that are invariant to zoom, shift, and rotation. These are often used in recognizing images with distortions such as fuzziness or random noise [20]. Hu moments are defined in Figure 1.

$$M_1 = \eta_{20} + \eta_{02} \;;$$
$$M_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2;$$
$$M_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 ;$$
$$M_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 ;$$
$$M_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})\left[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2\right] +$$
$$+(3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03}) \left[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2\right] \;; \quad (5)$$
$$M_6 = (\eta_{20} - \eta_{02})\left[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2\right] +$$
$$+4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03});$$
$$M_7 = (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12})\left[(\eta_{30} + \eta_{12}) - 3(\eta_{21} + \eta_{03})^2\right] +$$
$$+(3\eta_{21} - \eta_{30})(\eta_{21} + \eta_{03})\left[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2\right] ;$$

Figure 1. Hu Moments Definition.

3.3.3 RGB vs HSV Color Histogram

Shuhua and Gaizhi [21] explain the importance of color as an image vision property and the HSV color space, which is the third image property used in the creation of the global image features for this project. HSV is an alternate expression of RGB and carries a simpler means of calculation and a clearer perception than RGB. Color is not sensitive to rotate, shift, or scale and the calculation is simple compared to that of several other invariant features. The HSV model defines color according to hue, saturation, and luminance, where hue is the color name, saturation is the color purity, and luminance is the brightness. In data mining and pattern recognition it doesn't matter if RGB is transformed to another space, but Shuhua and Gaizhi's [21] results show that property preservation is better with RGB.

The images generated by Minor [9] during the visualization process were done so using two different layouts. The two layouts were used for both the malicious and the benign files, creating four sets of images: two sets of malicious and two sets of benign. The first layout preserved spatial locality by mapping bytes in a linear fashion going back and forth between left to right and right to left allowing bytes that were originally beside each other to stay that way for the most part. The second layout did not preserve spatial locality and mapped the bytes only from left to right. There were also two additional sets of images that were created without spatial locality preservation from repackaged versions of the applications. One set was from the malicious applications and the other set was from the benign applications.

3.4 Description of Experiments

Global features consisting of Hu Moments, Haralick Texture, and a Color Histogram were extracted from a total of 32,190 malicious images and the 2963 benign images provided by Minor [9] and fed to each of the seven machine learning algorithms listed in Table 1. This input is to train the algorithms to classify new images created from both malicious and benign applications as malware or safe. The training and testing experiments were broken down into the different layouts that they created and each of these subsets were of malicious images were used against two subsets of the benign images. There were also sets of images, both benign and malicious, that were created from repackaged applications and were used to train the machine learning algorithms to detect whether or not an application has been repackaged. The malicious subsets were composed of 13859 infected images with an only left to right layout that did not preserve the special locality of the bytes, 13859 infected images that followed a zig-zag pattern which did preserve spatial locality, and 2490 grey scale images that followed the zig-zag pattern and preserved spatial locality. The benign image subsets were composed of 2454 images with an only left to right layout that did not preserve the special locality of the bytes and 2454 images that followed the zig-zag pattern and preserved the spatial locality. The repackaged subsets were composed of 1982 infected repackaged images and 509 benign repackaged images, both of which followed the zig-zag pattern and preserving the spatial locality of the bytes. The ten-fold cross validation is performed on the training data to measure the effectiveness/accuracy of each of those seven algorithms and the three most accurate ones are considered for future experiments.

Table 2 represents the size of each data sets.

Table 2. Size of Each Data Set

Infected BMP (Grey)	Infected BMP2 RGB	Infected Snake (locality RGB)	Benign BMP (Grey)	Benign Snake (locality RGB)	Infected Repackaged (RGB)	Benign Repackaged (RGB)
13859	2490	13859	2454	2454	1982	509

Table 3. Training and Testing Combinations for Repackaging Detection

	Data Set 1	Data Set 2
Combination 1	Infected BMP (Grey)	Infected Repackaged (RGB)
Combination 2	Infected BMP (Grey)	Benign Repackaged (RGB)
Combination 3	Infected BMP2 (RGB)	Infected Repackaged (RGB)
Combination 4	Infected BMP2 (RGB)	Benign Repackaged (RGB)
Combination 5	Infected Snake (locality RGB)	Infected Repackaged (RGB)
Combination 6	Infected Snake (locality RGB)	Benign Repackaged (RGB)
Combination 7	Benign BMP (Grey)	Infected Repackaged (RGB)
Combination 8	Benign BMP (Grey)	Benign Repackaged (RGB)
Combination 9	Benign Snake (locality RGB)	Infected Repackaged (RGB)
Combination 10	Benign Snake (locality RGB)	Benign Repackaged (RGB)

In effort of achieving a success rate above 85% in using the machine learning algorithms to accurately classify repackaged images, several combinations were used as parameters for the training and testing of these algorithms. Each of the non-repackaged datasets were used against each of the repackaged datasets to determine which ones

yielded a result that was better than random (if any), and out of the ones that were better than random, which ones showed exceptional results. This created ten combinations to be used. Five containing the infected repackaged images, and five containing the benign repackaged images. The combinations are illustrated Table 3.

This chapter discussed the tools and features used including SKLearn, PIL, OpenCV, Haralick Texture, Hu Moments, and Color Histogram. This chapter also discussed the methodology, data, and training and testing combinations in this research. Chapter IV discusses the results of the experiments.

CHAPTER IV

RESULTS

In this chapter, the results of the experiments are reported. The first experiment results discussed are the two experiments replicating the Unver and Bakour global image feature experiment [8]. The second set of experiment results is all repackaging detection. The images used were pre-created by Minor [9] and consist of images that are in greyscale and images that use the RGB color space. Some of the images were created with a left to right byte layout while some were created with a zig zagged layout to preserve spatial locality.

4.1 Replication Experiments

The preliminary first experiment used images that were created from scratch out of the applications that Minor [9] provided. These applications are executable and are composed of DEX files extracted from the executables with Python's zipfile module. The DEX files are extracted, placed in their corresponding classification folder, and read in as binary data. Each binary data is then mapped to an image in the RGB color space using PIL and stored into its corresponding folder. After these steps are completed, a folder exists containing images from the infected applications and a folder containing images

from the uninfected applications. Python's OpenCV module is used to extract Haralick texture, Hu Moments, and the Color Histogram from each image which are concatenated together to form a single global feature.

The global feature from each image in every category is input into seven different machine learning classification algorithms using SKLearn in an effort to training the algorithm to accurately classify the infected and uninfected images. Once the algorithms are trained, ten-fold cross validation is performed. At the completion of the cross validation, a performance histogram containing the accuracy of all seven algorithms is displayed.

Images were created from 3394 DEX files extracted from the malicious applications. Additionally, images were created from 2451 DEX files extracted from the benign applications. These images were used in an effort to exceed Minor's 75% accuracy rate [9] and recreating the result of 98% accuracy achieved by Unver and Bakour [8].

Support Vector Machine was the most accurate of the seven with an 85.03% classification accuracy rate. Logistic Regression was the second most accurate yielding an accuracy rate of 84.99%. Linear Discriminant Analysis was third showing an accuracy rate of 84.63%. These are the three primary algorithms that should be the focus of future experiments if this research work is extended. This leads to the conclusion that the local features used by Unver and Bakour are required to receive a favorable accuracy and leads to the second experiment involving the detection of maliciousness. One observation made is the possibility that the machine learning algorithms defaulted to the

random probability when unbalanced training and testing data is involved. Figure 2 is a boxplot graph representing the results of all seven from the first preliminary experiment.

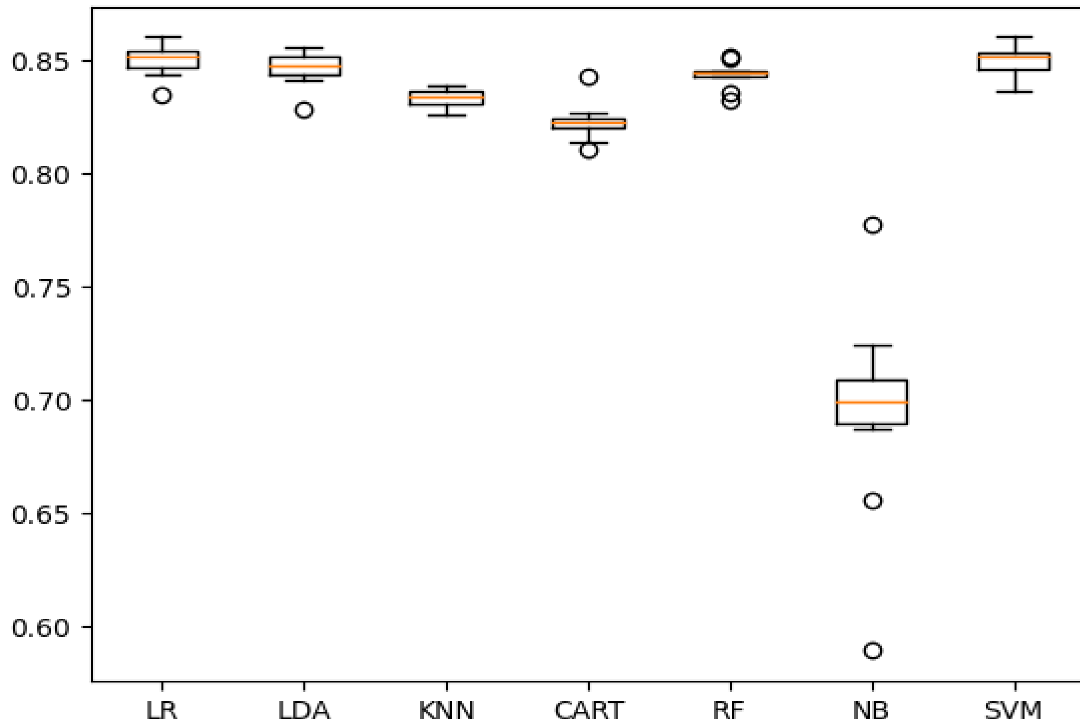

Figure 2. Results of First Malware Detection Experiment.

The second replication experiment with the pre-created images that Minor provided yielded results that did provide an accuracy rate was better than random. This experiment consisted of 2454 benign images created in grey scale, and 2490 infected images created in grey scale, neither of which preserved spatial locality. These images were created strictly with a left to right byte layout. K-Nearest Neighbor, Random Forest, and Linear Discriminant Analysis were the three best performers in this

experiment. K-Nearest Neighbor accurately classified 67.8% of the images, Random Forest accurately classified 67.6% of the images, and Linear Discriminant Analysis accurately classified 62.1% of the images. With K-Nearest Neighbor being the best, it was about seven percent lower than what Minor documented and far less than the 98% that we hoped to recreate. The wide range of accuracies and consistent outliers are noticeable as well. The important conclusion in this experiment is that the local features and training sample size make a big difference in the accuracy of malware classification. Figure 3 illustrates the results of this experiment.

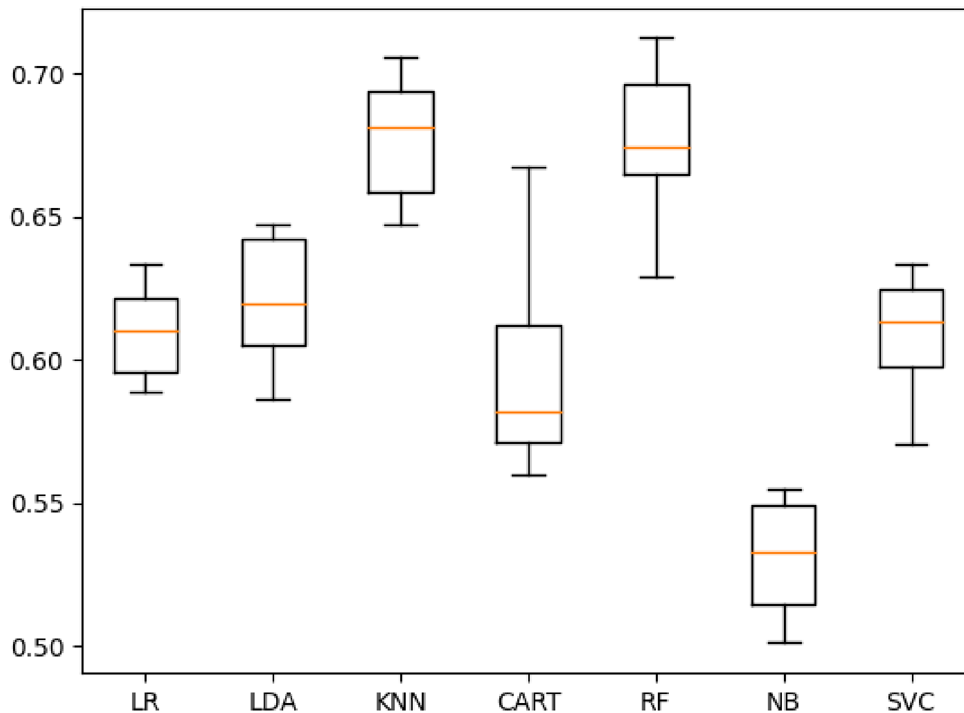

Figure 3. Results of Second Malware Identification Experiment.

4.2 Repackaging Results

The third experiment used combination eight. Random Forest, Support Vector Machine, and Linear Discriminant Analysis proved to be the most effective algorithms in this experiment. Random Forest accurately classified 99.4% percent of the images, Support Vector Machine accurately classified 99.2% percent of the images, and Linear Discriminant Analysis accurately classified 98.9% percent of the images. Note the significant difference in performance from Naïve Bayes. Figure 4 illustrates the results of this experiment.

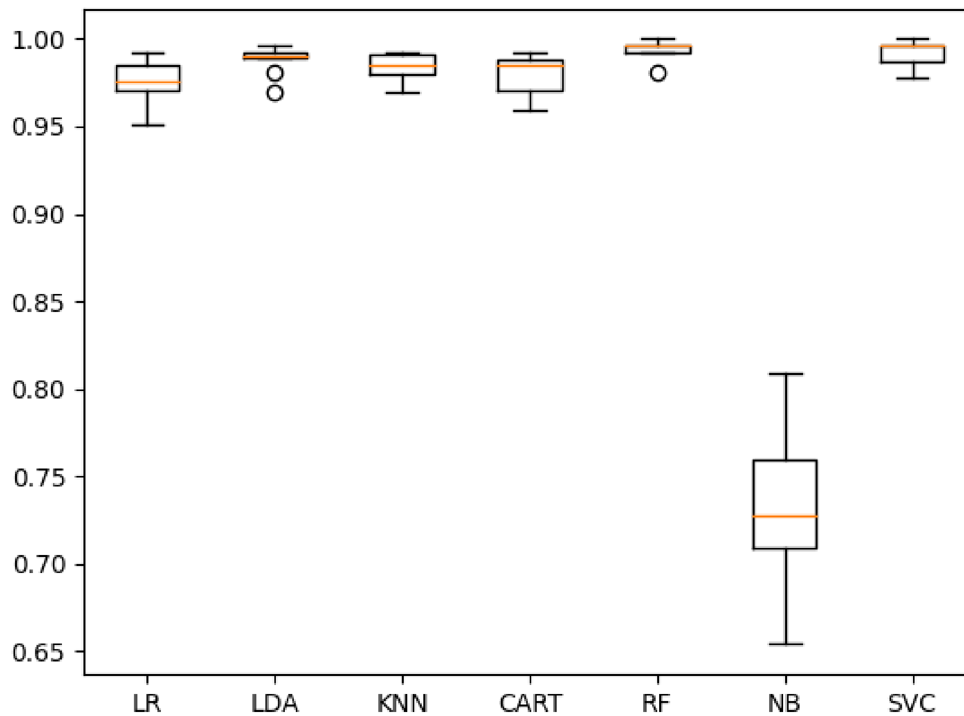

Figure 4. Results of First Repackaged Application Detection Experiment.

The fourth experiment used combination four. Random Forest, Support Vector Machine, and Linear Discriminant Analysis proved to be the most effective algorithms in this experiment. Random Forest accurately classified 99.5% percent of the images, Support Vector Machine accurately classified 99.4% percent of the images, and Linear Discriminant Analysis accurately classified 99.3% percent of the images. Note the significant difference in performance from Naïve Bayes. Figure 5 illustrates the results of this experiment.

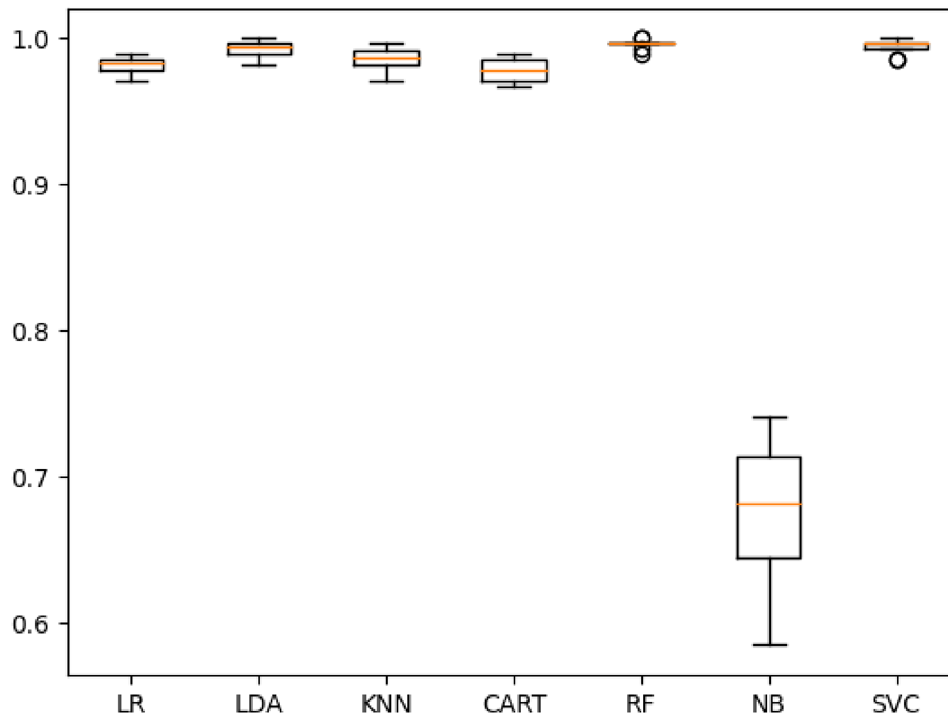

Figure 5. Results of Second Repackaged Application Detection Experiment.

The fifth experiment used combination one. These images were created strictly with a left to right byte layout. Random Forest, Support Vector Machine, and K-Nearest Neighbor proved to be the most effective algorithms in this experiment. Random Forest accurately classified 99.9% percent of the images, Support Vector Machine accurately classified 99.8% percent of the images, and K-Nearest Neighbor accurately classified 99.7% percent of the images. Note the significant difference in performance from Naïve Bayes. Figure 6 illustrates the results of this experiment.

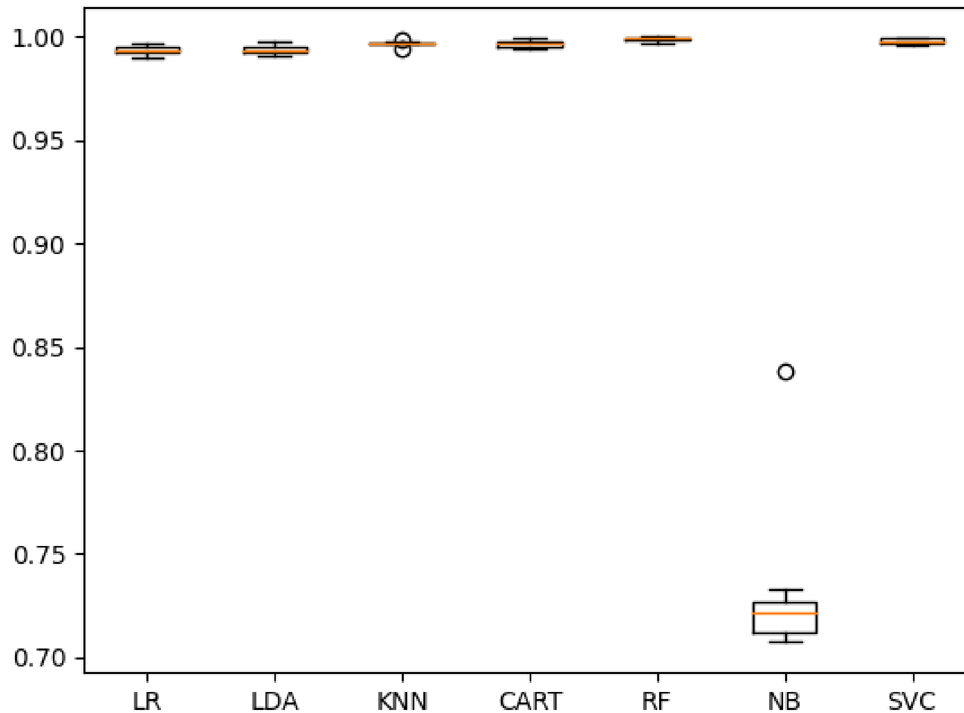

Figure 6. Results of Third Repackaged Application Detection Experiment.

The sixth experiment used combination three. Random Forest accurately classified 99.8% percent of the images, Support Vector Machine accurately classified 99.2% percent of the images, and K-Nearest Neighbor accurately classified 99.1% percent of the images. There was a significant difference in performance from Naïve Bayes. Figure 7 illustrates the results of this experiment.

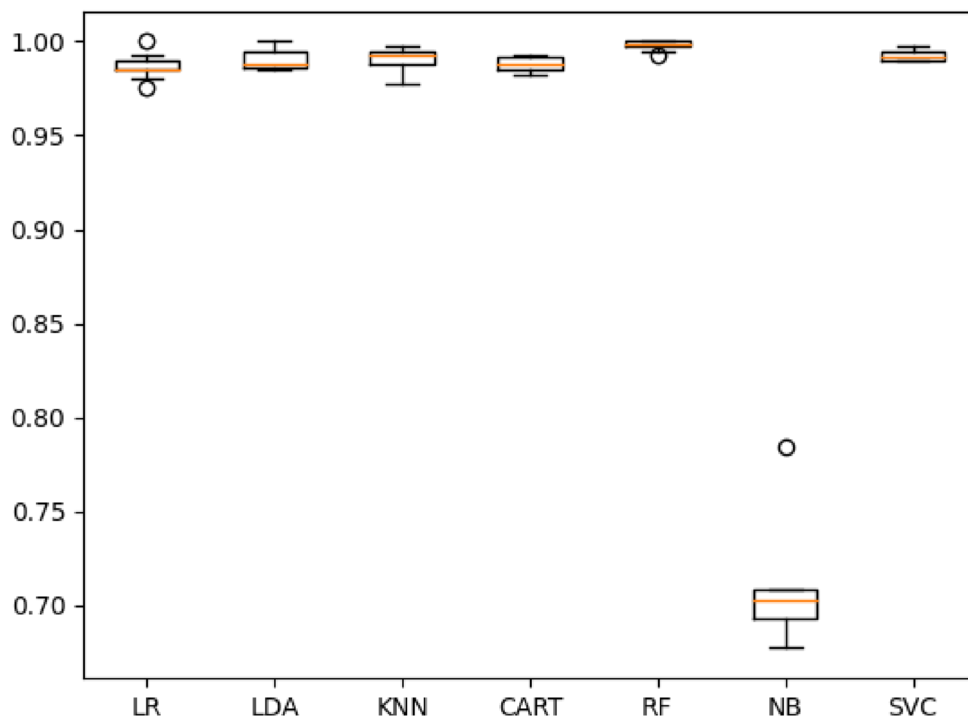

Figure 7. Results of Fourth Repackaged Application Detection Experiment.

The seventh experiment used combination seven. Random Forest accurately classified 99.7% percent of the images, Support Vector Machine accurately classified 99.4% percent of the images, and K-Nearest Neighbor accurately classified 99.2%

percent of the images. There was a significant difference in performance from Naïve Bayes. Figure 8 illustrates the results of this experiment.

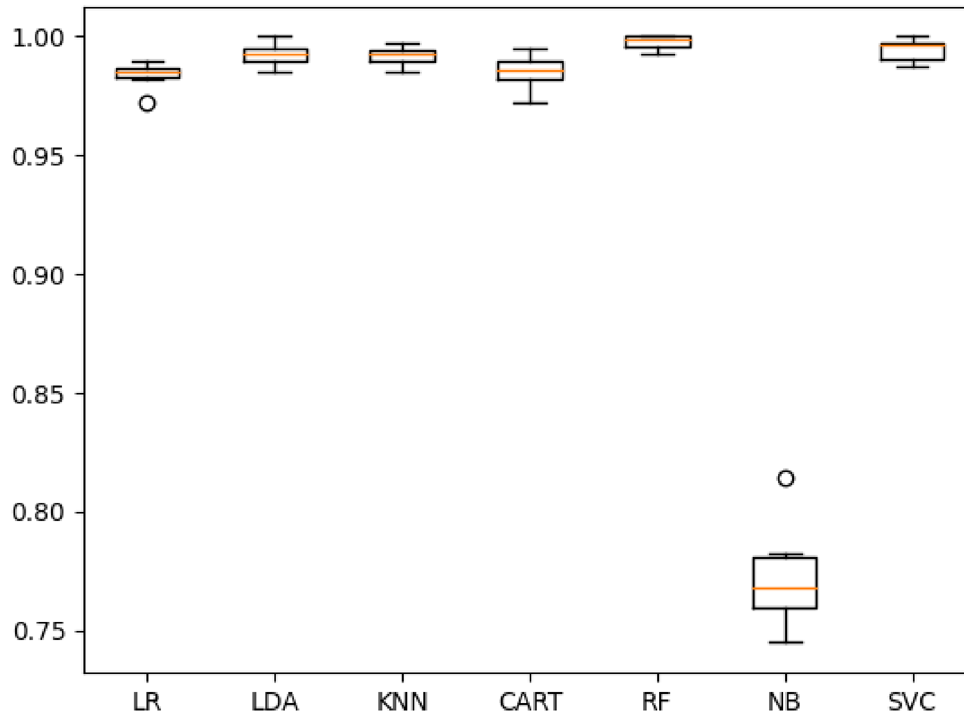

Figure 8. Results of Fifth Repackaged Application Detection Experiment.

The eighth experiment used combination two. Random Forest accurately classified 99.9% percent of the images, Support Vector Machine accurately classified 99.7% percent of the images, and Linear Discriminant Analysis accurately classified 99.7% percent of the images. There was a significant difference in performance from Naïve Bayes. Figure 9 illustrates the results of this experiment.

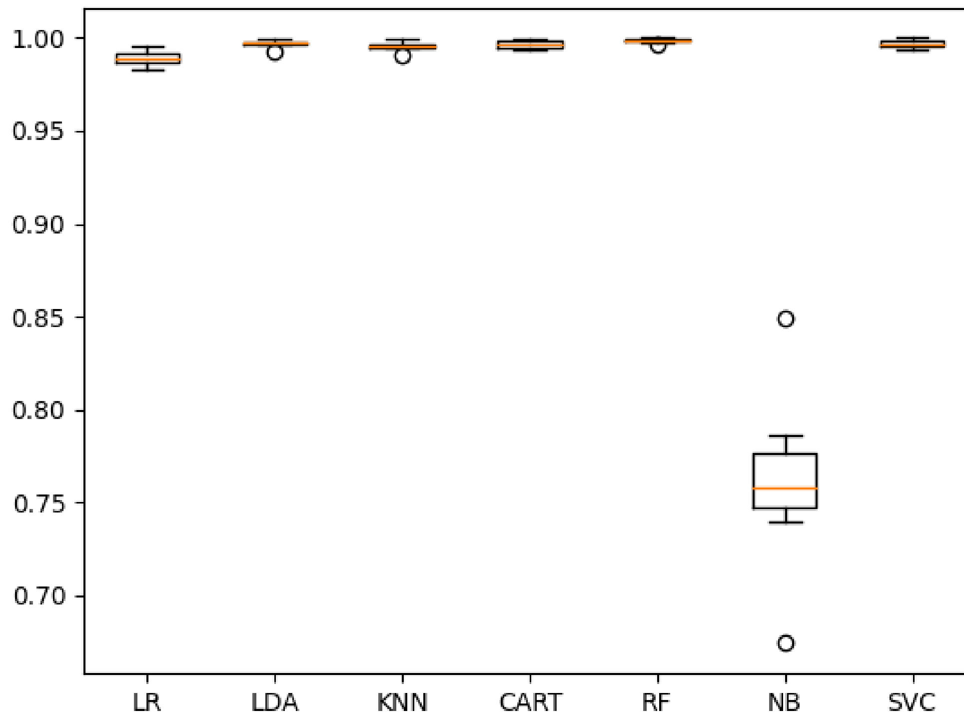

Figure 9. Results of Sixth Repackaged Application Detection Experiment.

The ninth experiment used combination five. Support Vector Machine accurately classified 87.4% percent of the images, Logistic Regression accurately classified 87.2% percent of the images, and Linear Discriminant Analysis accurately classified 86.6% percent of the images. There was a significant difference in performance from Naïve Bayes. Figure 10 illustrates the results of this experiment.

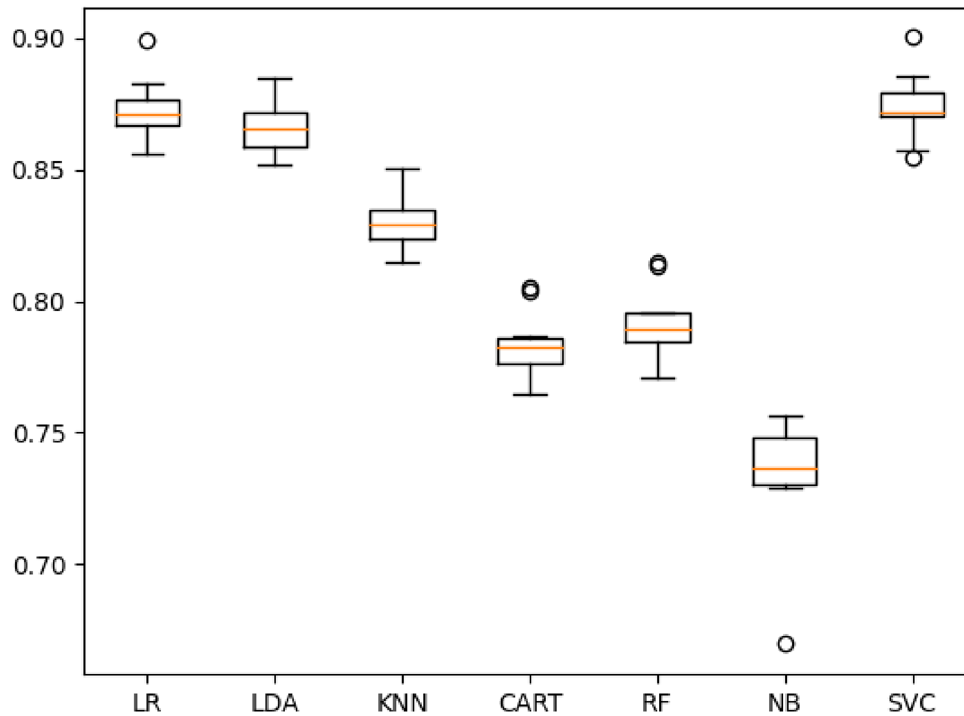

Figure 10. Results of Seventh Repackaged Application Detection Experiment.

The tenth experiment used combination six. Support Vector Machine accurately classified 96.5% percent of the images, Logistic Regression accurately classified 96.5% percent of the images, and Random Forest accurately classified 96.4% percent of the images. Figure 11 illustrates the results of this experiment.

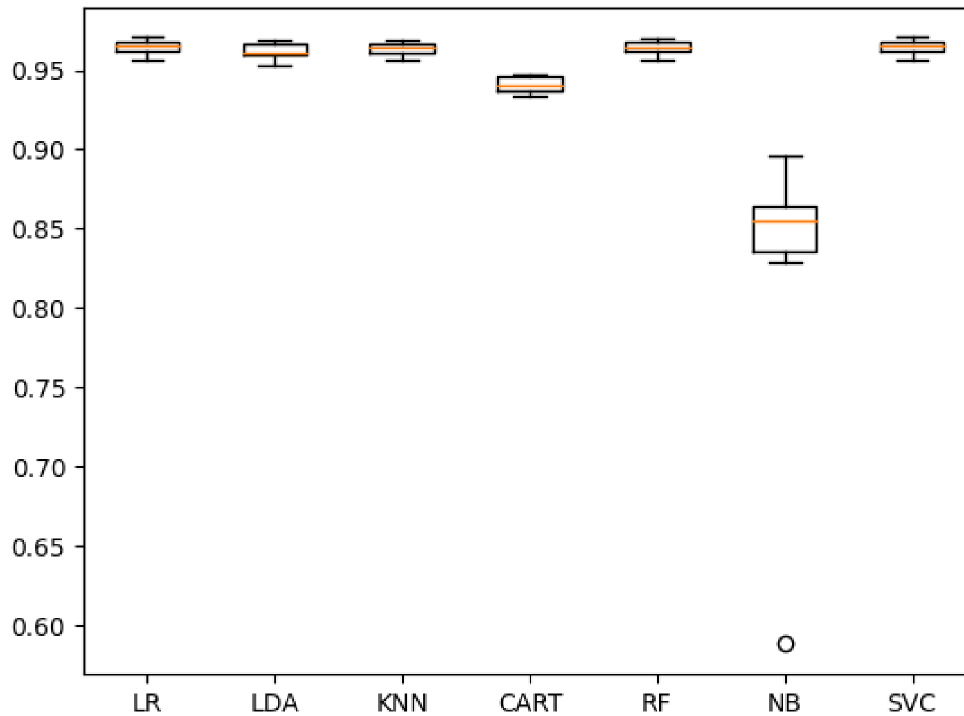

Figure 11. Results of Eighth Repackaged Application Detection Experiment.

The eleventh experiment used combination nine. Random Forest accurately classified 82.3% percent of the images, Classification and Regression Trees accurately classified 73.8% percent of the images, and K-Nearest Neighbor accurately classified 73.8% percent of the images. Figure 12 illustrates the results of this experiment.

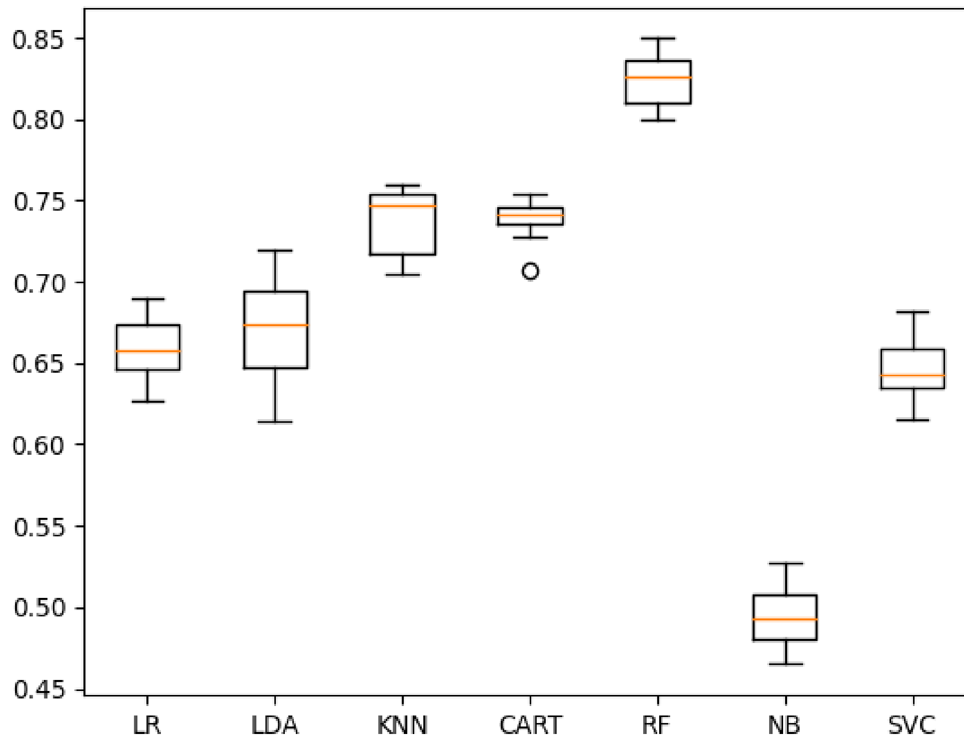

Figure 12. Results of Ninth Repackaged Application Detection Experiment.

The twelfth experiment used combination ten. Support Vector Machine accurately classified 82.9% percent of the images, Logistic Regression accurately classified 82.9% percent of the images, and Linear Discriminant Analysis accurately classified 82.3% percent of the images. There was a significant difference in performance from Naïve Bayes. Figure 13 illustrates the results of this experiment.

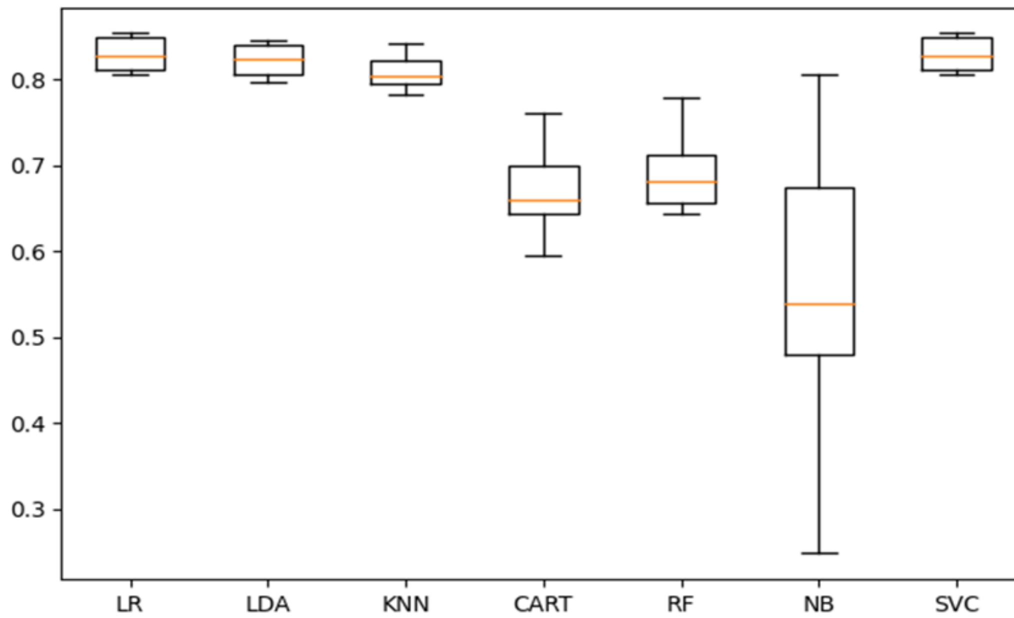

Figure 13. Results of Tenth Repackaged Application Detection Experiment.

This chapter discussed the types of experiments conducted and the results of each. Chapter V discusses conclusions, importance of this research, significance of the results, and possible future work.

CHAPTER V

CONCLUSIONS

5.1 Importance

Year over year, the number of smart phone users has increased ever since the very first smart phone was released. Today, over 35% of the world's population owns a smartphone. Figure 14 illustrates both the current and projected growth, with growth not anticipated to slow down any time soon [22]. The Android operating system has been the leader in worldwide mobile operating system usage for over 70% of users [22]. With such growth and usage, and the way these devices are relied on for many personal and private activities, Android devices are a prime target for attackers. The attacks range anywhere from pirating applications for profit, removing ads, redirecting legitimate ad revenue, obtaining paid services for free, and identity theft.

Today owners use smartphones to pull up menus at restaurants, control household appliances and heating and air conditioning units, look up maps, use GPS services, store pictures and songs, watch movies, keep track of appointments and meetings, and the list goes on and on. Smart phones store a huge amount of direct personal information (banking, pictures of documents, etc.) and indirect information about our everyday lives (likes, interests, habits, etc.) that could be leveraged by attackers. At the rate that new

applications are being deployed in the markets, the possible uses of smartphones are increasing exponentially, and in turn, both the application screening responsibilities of the markets and the opportunities for attackers to repackage applications are drastically increasing [12]. This research could lead to an increase in our ability to detect malicious and/or pirated Android applications and drastically reduce attacks [22].

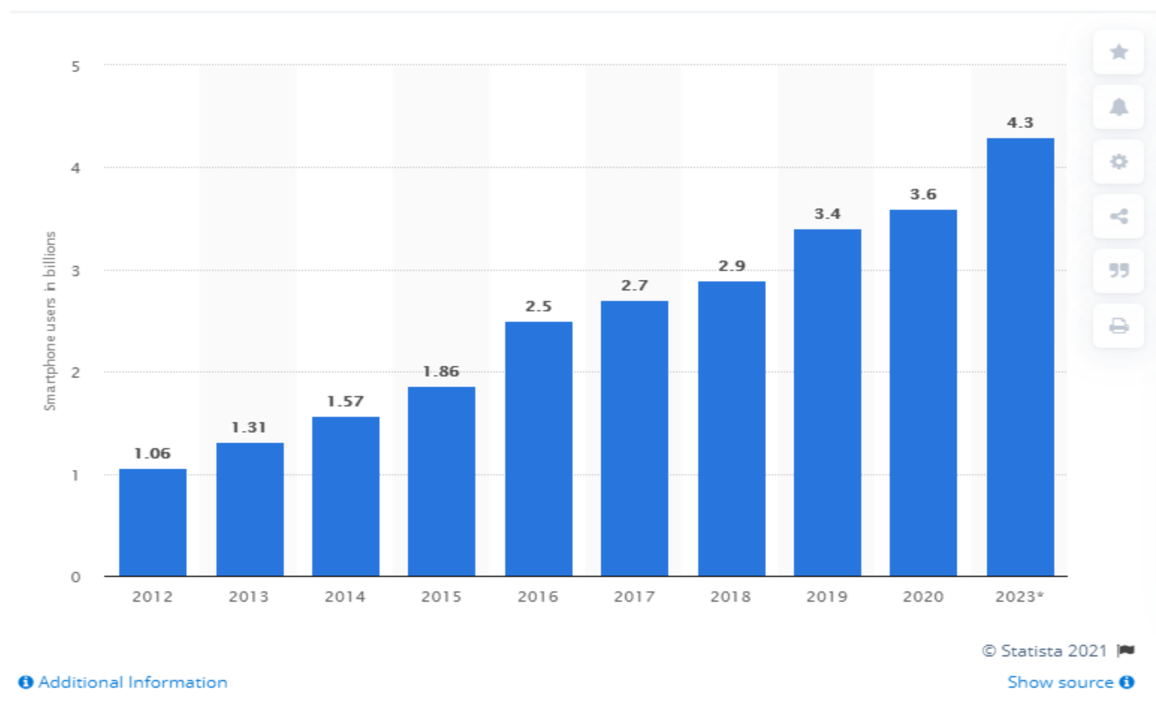

Figure 14. Past and Projected Growth of Smartphone Users (In Billions).

5.2 Significance of Results

A few promising results were identified in this research. Experiment number three that contained 509 images from uninfected repackaged applications and 2454 uninfected original applications yielded above 99% classification accuracy. Experiment number four that contained 509 images created from uninfected repackaged applications and 2490 images from infected original applications yielded above 99% classification accuracy. Experiment number five that contained 13859 images created from infected original applications and 1982 images created from infected repackaged applications yielded above 99% classification accuracy. Experiment number six that contained 2490 images from infected original applications and 1982 images created from infected repackaged applications yielded above 99% classification accuracy. Experiment number seven that contained 2454 images created from uninfected original applications and 1982 images created from infected repackaged applications yielded above 99% classification accuracy. It should be noted that attempting to preserve the spatial locality rather than placing the bytes strictly from left to right when mapping the image yields less favorable results on the identical experiment. These experiments should be executed on a more recent and larger data set in the future. A summary of the results of the repackaging detection is displayed in Figure 15.

B	C	D	E	F	G	H	I	J	K	L
Uninfected	Uninfected Zig Zag RBG	Uninfected Repackaged	Infected	Infected	Infected Zig Zag RGB	Infected Repackaged	1st Place	2nd Place	3rd Place	Better than Random?
2454			13859				LR 84.9	LDA 84.6	RF 84.3	N
2454				2490			KNN 67.8	RF 67.6	LDA 62.1	Y
2454		509					RF 99.4	SVC 99.2	LDA 98.9	Y
		509		2490			RF 99.5	SVC 99.4	LDA99.3	Y
			13859			1982	RF 99.9	SVC 99.8	KNN 99.7	Y
				2490		1982	RF 99.8	SVC 99.2	LDA 99.1	Y
2454						1982	RF 99.7	SVC 99.4	KNN 99.2	Y
		509	13859				RF 99.9	SVC 99.7	LDA 99.7	Y
					13859	1982	SVC 87.4	LR 87.2	LDA 86.6	N
		509			13859		SVC 96.5	LR 96.5	RF 96.4	N
	2454					1982	RF 82.3	CART 73.8	KNN 73.8	Y
	2454	509					SVC 82.9	LR 82.9	LDA 82.3	N

Figure 15. Summary of Repackaging Detection Results.

This chapter discussed the conclusions of this research and promising results that could lead to future work. This chapter also discussed the significance of the results and why this research is important.

REFERENCES

REFERENCES

[1] Nguyen, T., McDonald, J.T., Glisson, W.B., & Andel, T.R. (2020). Detecting Repackaged Android Applications Using Perceptual Hashing. *Hawaii International Conference on Systems Sciences (HICSS)*, pp. 6641-6650.

[2] Jordan, M. I., and Mitchell, T. M. (2015, July 17). Machine learning: Trends, perspectives, and prospects. *Science*, *349*(6245).

[3] (N.A.). (2011). Google's Android Becomes the World's Leading Smart Phone Platform. *Canalys.* Retrieved From https://www.canalys.com/newsroom/google%E2%80%99s-android-becomes-world%E2%80%99s-leading-smart-phone-platform.

[4] Kartel, A., Novikova, E., and Volosiuk, A., "Analysis of Visualization Techniques for Malware Detection," *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, St. Petersburg and Moscow, Russia, 2020, pp. 337-340, doi: 10.1109/EIConRus49466.2020.9038910.

[5] Venkatraman, S. and Alazab, M., "Classification of Malware Using Visualisation of Similarity Matrices," 2017 Cybersecurity and Cyberforensics Conference (CCC), London, UK, 2017, pp. 3-8, doi: 10.1109/CCC.2017.11.

[6] Chen, H., Du, R., Liu, Z., and Zu, H. (2018). *Android Malware Classification using XGBoost based on Images Patterns*. Retrieved March 18, 2021, from IEEE.

[7] Wang, W., Gao, Z., Zhao, M., Li, Y., Liu, J., and Zhang, X. "DroidEnsemble: Detecting Android Malicious Applications with Ensemble of String and Structural Static Features," in IEEE Access, vol. 6, pp. 31798-31807, 2018, doi: 10.1109/ACCESS.2018.2835654.

[8] Ünver, H., and Bakour, K. "Android malware detection based on image-based features and machine learning techniques," in SN Applied Sciences, vol. 2**,** 15 pages, 2020, doi: 10.1007/s42452-020-3132-2

[9] Minor, A. (2019). *Android Malware Detection Using Textural Feature Recognition of Visualized Binary* (Publication Number 13859428) [Master's thesis, University of South Alabama]. ProQuest Dissertations and Theses Global.

[10] Jung, J., Lim, K., Kim, B., Cho, J., Han, S., and Suh, K. "Detecting Malicious Android Apps using the Popularity and Relations of APIs," 2019 IEEE Second International Conference on Artificial Intelligence and Knowledge Engineering (AIKE), 2019, pp. 309-312, doi: 10.1109/AIKE.2019.00062.

[11] Rastogi, S., Bhushan, K., and Gupta, B. B. (2016, April 3). *Android applications repackaging detection techniques for smartphone devices*. Procedia Computer Science. https://www.sciencedirect.com/science/article/pii/S1877050916000089#:~:text=Repackaged%20apps%20are%20usually%20infected,repackage%20and%20re-

lease%20the%20app.&text=There%20are%20many%20techniques%20which%20focus%20entirely%20on%20detecting%20repackaged%20apps.

[12] Song, L., Tang, Z., Li, Z., Gong, X., Chen, X., Fang, D., and Wang, Z., "AppIS: Protect Android Apps Against Runtime Repackaging Attacks," 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS), 2017, pp. 25-32, doi: 10.1109/ICPADS.2017.00015.

[13] Chen, Q., Wang, J., and Wang, Y., "An Online Approach for Detecting Repackaged Android Applications Based on Multi-user Collaboration," 2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), 2015, pp. 312-315, doi: 10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.66.

[14] Sun, X., Han, J., Dai, H., and Li, Q., "An Active Android Application Repacking Detection Approach," 2018 10th International Conference on Communication Software and Networks (ICCSN), 2018, pp. 493-496, doi: 10.1109/ICCSN.2018.8488263.

[15] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, É., "Scikit-learn: Machine Learning in Python", Journal of Machine Learning Research, vol. 12, pp. 2825-2830, 2011.

[16] Lundh, F., (n.d.). Pillow. Retrieved May 22, 2021, from https://pillow.readthedocs.io/en/stable/.

[17] Aggarwal, C. C. (2015). *Data mining: The textbook*. Cham Switzerland: Springer.

[18] *About*. OpenCV. (2020, November 4). Retrieved January 15, 2021, from https://opencv.org/about/.

[19] Haralick, R., Shanmugam, K., and Dinstein, I., "Textural Features for Image Classification," in IEEE Transactions on Systems, Man, and Cybernetics, vol. SMC-3, no. 6, pp. 610-621, Nov. 1973, doi: 10.1109/TSMC.1973.4309314.

[20] Peleshko, D., Peleshko, M., Kustra, N., and Izonin, I., "Analysis of invariant moments in tasks image processing," 2011 11th International Conference The Experience of Designing and Application of CAD Systems in Microelectronics (CADSM), Polyana, Ukraine, 2011, pp. 263-264.

[21] Shuhua. L., and Gaizhi, G., "The application of improved HSV color space model in image processing," 2010 2nd International Conference on Future Computer and Communication, Wuhan, China, 2010, pp. V2-10-V2-13, doi: 10.1109/ICFCC.2010.5497299.

[22] O'Dea, S., (2021, March 18). Smartphone users 2020. Retrieved March 30, 2021, from https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.

BIOGRAPHICAL SKETCH

BIOGRAPHICAL SKETCH

Name of Author: Robert Cox

Graduate and Undergraduate Schools Attended:

The University of South Alabama, Mobile, Alabama

Degrees Awarded:

Master of Science in Computer and Information Sciences, 2021, Mobile, Alabama

Bachelor of Science in Computer Science, 2019, Mobile, Alabama

Awards and Honors:

Scholarship for Service, National Science Foundation, 2019–2021

Research Experience for Undergraduates, National Science Foundation, 2018 – 2019

Computing Undergraduate Research Experience, School of Computing, 2018