إقـــرار

أنا الموقع أدناه مقدم الرسالة التي تحمل العنوان:

كشف البرمجيات الخبيثة في تطبيقات أنظمة الأندرويد باستخدام خوارمية المتشابهات

أقر أن ما اشتملت عليه هذه الرسالة إنما هو نتاج جهدي الخاص، باستثناء ما تمت الاشارة إليه حيثما ورد، و إن هذه الرسالة ككل أو أي جزء منها لم يقدم من قبل لنيل درجة أو لقب علمي أو بحثي لدى أي مؤسسة تعليمية أو بحثية أخرى.

Declaration

The work provided in this thesis, unless otherwise referenced, is the researcher's own work, and not has been submitted elsewhere for any other degree or qualification.

Student's name:
Signature:
Date:6/9/2014

اسم الطالب: محمود زهير سعيد الكردي
التوقيع: [illegible]
التاريخ:6/9/2014

The Islamic University of Gaza

Faculty of Engineering

Computer Engineering Department

كشف البرمجيات الخبيثة في تطبيقات أنظمة الأندرويد باستخدام خوارمية المتشابهات

Malware Detection for Android Applications Using SimHash Algorithm

By

Mahmoud Zuher Alkurdi

A Master of Science Thesis

Supervisors:

Dr.Aiman Abu Samra

Dr.Hasan Qunoo

June,2014

بسم الله الرحمن الرحيم

الرقم ج س غ/35/ Ref

التاريخ 2014/06/25 Date

نتيجة الحكم على أطروحة ماجستير

بناءً على موافقة شئون البحث العلمي والدراسات العليا بالجامعة الإسلامية بغزة على تشكيل لجنة الحكم على أطروحة الباحث/ **محمود زهير سعيد الكردي** لنيل درجة الماجستير في **كلية *الهندسة*** قسم **هندسة الحاسوب** وموضوعها:

كشف البرمجيات الخبيثة في تطبيقات أنظمة الأندرويد باستخدام خوارمية المتشابهات
Malware Detection for Android Applications Using SimHash Algorithm

وبعد المناقشة التي تمت اليوم الأربعاء 27 شعبان 1435هـ، الموافق 2014/06/25م الساعة الثانية مساءً، اجتمعت لجنة الحكم على الأطروحة والمكونة من:

د. أيمن أحمد أبو سمرة	**مشرفاً ورئيساً**	[illegible]
د. حسن نجيب قنوع	**مشرفاً**	[illegible]
أ.د. محمد أمين مكي	**مناقشاً داخلياً**	[illegible]
د. أحمد فؤاد عبد العال	**مناقشاً خارجيًا**	[illegible]

وبعد المداولة أوصت اللجنة بمنح الباحث درجة الماجستير في **كلية *الهندسة*/** قسم **هندسة الحاسوب**.

واللجنة إذ تمنحه هذه الدرجة فإنها توصيه بتقوى الله ولزوم طاعته وأن يسخر علمه في خدمة دينه ووطنه.

والله ولي التوفيق،،،

مساعد نائب الرئيس للبحث العلمي والدراسات العليا

[illegible]

أ.د. فؤاد علي العاجز

شئون البحث العلمي والدراسات العليا Research & Graduate Affairs

الملخص

تحليل أكواد تطبيقات أنظمة الأندرويد وكشف الخبيث منها يعتبر محور بحث هام جدا ، هناك الكثير من الأبحاث لابتكار طرق جديدة وخلاقة لعمل ذلك بطريقة سهلة ودقيقة مع الأخذ بعين الاعتبار التكلفة الحوسبية، عدد هذه الأبحاث ينمو بسرعة كبيرة وذلك لتزايد عدد تطبيقات انظمة الأندرويد بشكل ضخم وكبير .

هذه الأطروحة تحاول أن تساهم في حل هذه المشكلة عن طريق أستخدام طريقة مبتكرة وحديثة لتميز الأكواد الخبيثة حيث تقوم بتحليل الأكواد ما قبل التنفيذ ومن ثم تقارن ما بين تشابهها مع قاعدة بيانات أكواد خبيثة مكتشفة سابقا عن طريق استخدام خوارزمية اسمها (SimHash).

قامت هذه الأطروحة بتطبيق هذه المفاهيم كجزء من برنامج حماية مشهور ويستخدم بشكل كبير ومعتمد لدى عدد من شركات برمجيات أمن البيانات اسمه (Androguard) وذلك للتأكد من صلاحية الفكرة وتم مقارنة النتائج لتحليل الأكواد بين الطريقة الحديثة و (Androguard) والمفاضلة بينهما من حيث دقة النتائج والتكلفة الحوسبية ,حيث تم استخدام عينة من البرمجيات الخبيثة تتكون من 130 تطبيق لأجراء هذه المقارنة وكانت النتيجة أن الطريقة المقترحة في الرسالة وفرت 70% من الوقت الذي تم استهلاكه في Androguard لتحليل واستخراج بصمات التطبيقات مع توزيع وقت مشابه وخطأ بسيط في دقة النتائج ، وهذا يرجع لبساطة وسهولة الطريقة المستخدمة من قبل الرسالة لتحليل الأكواد واستخراج البصمات.

.

Abstract

Code analysis and Malwares detection for Android applications are considered as an serious problem; there are many researches to apply new and creative techniques that can detect Malwares with low computational cost. These researches are being rapidly grown because of wide using and a huge number of new applications.

This thesis tries to attack this problem by presenting a new and creative method to analyze static code and measure similarity with available dataset of known Malwares applications using (SimHash) algorithm.

Thesis immigrate its idea to a well-known and wide officially used antivirus project, which is considered as a part of famous antiviruses programs and called (Androguard). We apply thesis idea in Androguard project to test its feasibility by making a comparison study between it and modified Androguard; we used Malware dataset of 130 Android applications for this research. As a result, proposed method saves about 70% of time with similar results and time distribution behavior in compare with original Androguard and little scarification in accuracy, this refer to simplicity of generating signatures and measuring similarity using SimHash algorithm.

Dedication

To my loving parents, family and to my fiancée (Alaa)

Acknowledgement

I would like to acknowledge my thesis supervisors Dr. Ayman Abu Samra and Dr. Hasan Qunoo for their guidance and valuable help; I would also to thank Dr.Moutaz Alazab in Deaking University Australia for his advices and his valuable discussion around my work. Finally I appreciate Ms. Milla Parkour researcher at Bugix–Security Research the owner of Contgiodump blog for giving me permission for her Android Malwares dataset.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Topic Area

Smartphones are used everywhere, by everyone, for all purposes, it is the latest technology trend of the 21st century. Today's social life requires us to stay in always connection with internet by smartphones, also smartphones are being rapidly integrated into enterprises, government agencies, and even the military, In fact smartphones are used by all people around the world from all ages and for various usage. All of these are reasons for the wide development of smartphones hardware and software.

Smartphones are based on several platforms; one of the most popular is Android. The popularity of Android has enabled the application marketplace to grow dramatically, the black market presence has also grown rapidly where paid applications are modified for free download and from untrusted websites or stores, smartphone user may exposed to various information security threats when he uses his phone, these threats can disrupt the operation of the smartphone, and transmit or modify the user data. For these reasons, Android applications must guarantee privacy and integrity of the information they handle.

There are several countermeasures and researches to guarantee privacy and integrity of apps by detecting and preventing Malware threat in mobile devices some of these are signature based antivirus scanners which efficiently detect known Malwares, others depends on detection and classification method in which they classify source code to detect Malware , even if it has no background of mobile applications.

These countermeasures and researches are different in their accuracy and mobile resources consumption and there are a lot of researches which try to solve these problems.

1.2 Research Question

Android platform is the most popular among Malwares designers. Because of Android Malwares fast growth, we need to develop effective solutions. General works to Android Malwares are currently focused on known antivirus scanners which efficiently detect known Malwares. [7]

Android Malware detecting using static analysis can provide a comprehensive view, it is still subjected to high cost in environment, So the question is how to detect unknown Malware by rephrased method without high computation cost??.

1.3 Significance

There is a rising danger associated with Malware applications at mobile devices, so the problem of detecting such Malwares is an interesting topic, Machine Learning based system is good idea for Malware detection on Android applications, this technique will be used on static features that are extracted from Android's application files. Each application in Android is packaged in an .apk archive which is similar to standard Java .jar files and comprise of both code and resources. Android .apk files encapsulate valuable information that can help in understanding applications behavior. The information can be collected by reverse engineering tools, it includes requested permissions, framework methods called up by the application, also extracts the information e.g., requested permissions, Intent messages passing, etc.) from each application's manifest file.

In this research a new model of Android Malware detection will be presented, this model depends on extracting features from .apk file by generating a Hash code (SimHash) for its reversed code and its Manifest .xml file, this Hash code will be used for similarity measurement to detect application behavior by comparing it with a dataset of Malwares applications, modified method were compared with well-known application used for the same purpose , called Androguard which is complete tool programmed by Virus Total group for antiviruses purposes, This tool uses multifunction for code analysis and features extraction and Malware detection . A comparison study has been done on a training data collected from Malware dataset. As a result research modified Androguard method saves 70% of time compared with Androguard tool with result similar to it. This refers to using SimHash instead of normal compression to measure similarity, and saving the time of analyzing reverse code.

1.4 Thesis Structure

This thesis is organized as follow:

Chapter 1; Introduction: In this chapter thesis provides an introduction about thesis problem, questions and significance, this chapter describes why we choose this title for thesis and the idea of proposed solution.

Chapter 2; Background and Related Work: This chapter provides a background about Android system, application and programming. It also talks about Malwares in general and Malware in Android applications, at the end of this chapter there is a group of related work in the same topic of this thesis.

Chapter 3; Research Approach and Tools: This chapter describes in theoretical view the most important used tools in this thesis; it provides readers with description about algorithms and used applications.

Chapter 4; 4 Methodology Evaluation and Analysis.: Here readers can show the used methodology for thesis, and how we prove the feasibility of our idea, this chapter also provides details about experiments and it results, in addition it provides more details about comparison study between thesis tools and Androguard.

Chapter 5; Conclusion and Future Work: A complete conclusion has been written in this chapter; also we talk about future work in related to his topic.

Chapter 6; References: this chapter is a list of all sources associated with thesis.

Chapter7; Appendices: In this chapter author attaches used codes and his modifications.

1.5 Summary

As a summary of this chapter we can say that today life depends on latest technologies which provide fast and available services for its users, smartphones are one of these technologies, it is used everywhere, by everyone, for all purposes. The wide use of smartphones applications leads for wide growth in Malwares applications which aims to threat users, This thesis tries to solve the problem of high cost Malware detection by providing a new method which saves computation cost by using SimHash algorithm .As a result research modified method save 70% of time comparing with compared (Androguard) tool with result similar to it. This refers to using SimHash instead of normal compression to measure distance and saving the time of analyzing reverse code.

2 Background and Related Work

The motivation of this chapter is to explain the concept of Android. This Chapter starts with, a short introduction into Android architecture and applications. In Section 2.1 a summarized description of Android operating system will be provided, Section 2.2 will talks about Android applications fundamentals, Section 2.3 is a brief summary of Android Malwares types and methods of detection, Section 2.4 summarizes some related work in the field of Malware detection tools. Finally section 2.5 is Summary for chapter.

2.1 Android System Architecture

Android is a new open source mobile platform that was designed by Google .Android applications utilized advanced hardware and software to bring benefits and value to its users.

The architecture of Android is implemented as a software stack, customized for mobile devices. Figure 2-1 shows some of the most important components of this stack [5].

The core of the Android platform is a Linux kernel. The kernel's responsible for handling device drivers, resource access, memory process, power management and other typical OS duties. The kernel also acts as an abstraction layer between the hardware and other software stack.

On top of the kernel are several native C/C++ libraries. Most of the application framework access these core libraries through the Dalvik Virtual Machine DVM, which can be seen as a gateway to the Android platform. This access is based on Java APIs that are thin wrapper classes around the native code using the Java Native Interface.

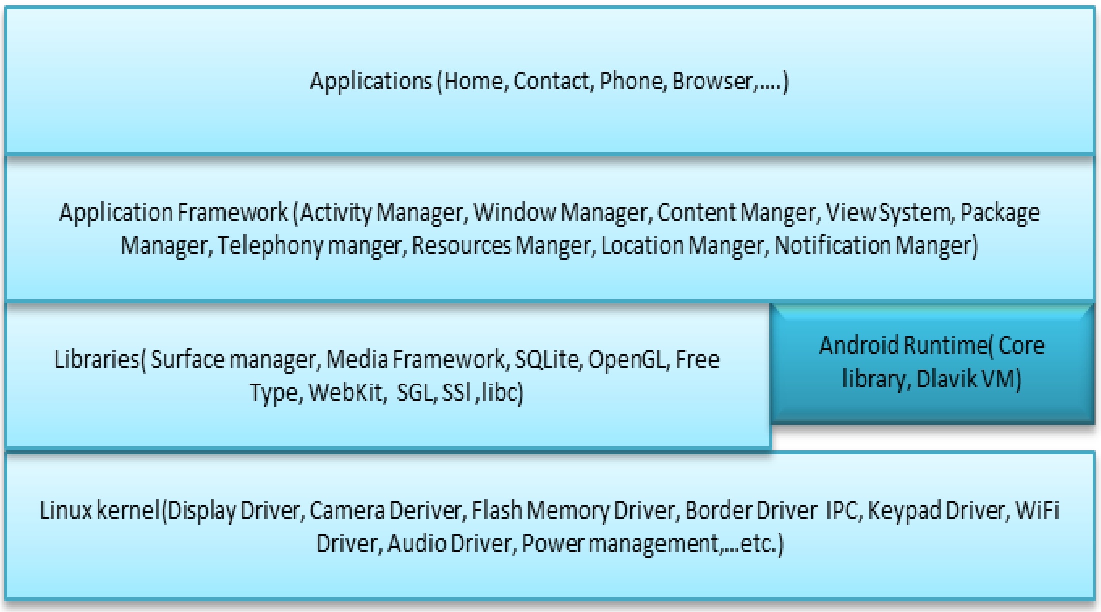

Applications (Home, Contact, Phone, Browser,....)
Application Framework (Activity Manager, Window Manager, Content Manger, View System, Package Manager, Telephony manger, Resources Manger, Location Manger, Notification Manger)
Libraries(Surface manager, Media Framework, SQLite, OpenGL, Free Type, WebKit, SGL, SSl ,libc)
Android Runtime(Core library, Dlavik VM)
Linux kernel(Display Driver, Camera Deriver, Flash Memory Driver, Border Driver IPC, Keypad Driver, WiFi Driver, Audio Driver, Power management,...etc.)

Figure 2-1: Android OS Architecture

Programmers develop end user applications on top of the main libraries in the application framework which provide access to resources management [3], features like.

- **Activity manager:** manages the lifecycle of an application as applications are started, suspended, resumed or destroyed see Figure 2-2 and Table1, This also provides a navigation stack for the graphical views as the user is navigating through the different views within an application.
- **Content providers:** enables applications to share its own data and access phone data such as contacts and SMS entries.
- **Resource manager:** provides access to resources outside code such as strings, layout XML files and graphics.
- **View system:** access to views that can be used to build the application and include buttons, lists, grids etc.

- **Telephony manager:** information about telephony services on the device. Applications can also use this manager to determine services and states and access some subscriber information. This also enables the possibility for applications to register as listeners to receive changes of the telephony state.
- **Package manager:** access information related to the packages installed on the device.
- **Location manager:** access to the system location services. The services allow applications to obtain information about geographical location.

The Android software development kit supports most of the Java SE except for the AWT and Swing UI components, thus making almost all Java SE libraries available compared to J2ME which is very stripped down. Included in the SDK is an emulator to run, debug and test end-user developed applications. The emulator acts like most of the features of a real device except some limitations regarding camera and video capture, headphones, battery simulation and Bluetooth. The emulator is based on Quick Emulator QEMU which enables several operating systems to be executed on one machine and under different architectures. The SDK also contains several tools to assist developers, the most significant is Android manages virtual devices AVDs, projects and installed components on a SDK.

The Android Virtual Device AVD is an emulator configuration that enables modeling of an actual device by defining hardware and software options that are then emulated. These options include: mapping to a system image, hardware features and dedicated storage area for simulating a SD card that contain user data. The system image contains the version specific Android implementation that includes the application framework and DVM.

2.2 Android Application Fundamentals

The four essential building blocks of an application are; Activities, Services, Content providers and Broadcast receivers [5].

- **Activity:** Represents a single screen in the user interface. Users implement this by sub classing the Activity class and implementing necessary lifecycle callbacks, see Figure 2-2 and Table 2-1.
- **Service**: A component that runs in the background that is not a user interface. A service is typically started by an Activity component but can be started by other components too.
- **Content provider:** Manages shared application data. This data is saved in the file system, SQLite3 database or other persistent storage location. Through this component other applications can access data to perform queries or make modifications if the content provider allows it. Content providers are accessed via Content Resolver objects. Other ways of storing data is using Shared Preferences that write a key, value pair to a XML file.

- **Broadcast receivers:** Each application must have an AndroidManifest.xml file which provides information about the application to the Android system. and sending

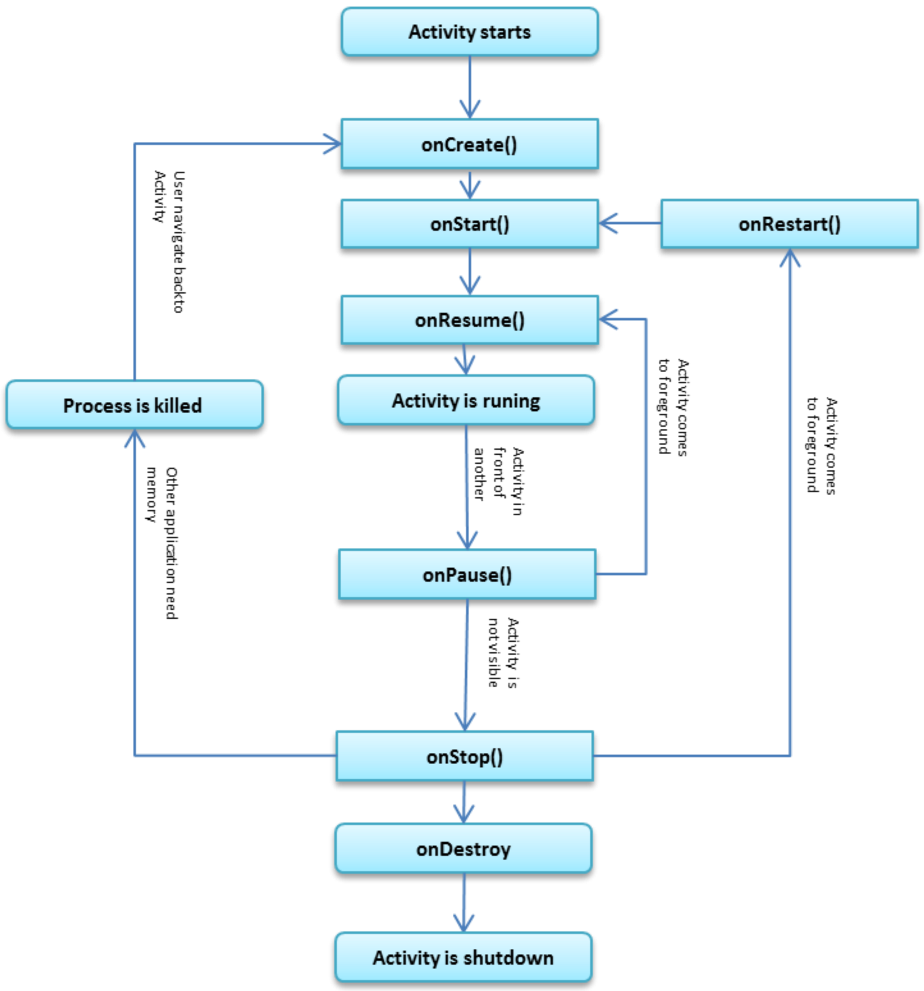


Figure 2-2: Lifecycle of an Activity and its Callback Method

broadcasts which are launched using asynchronous messages, Intents. Intent is an abstract description of an operation to perform. These messages contain information about action and data to operate on.

Table 2-1: Lifecycle of an Activity and its Callback Method Describtion

Method	Description	Killable?	Next
onCreate()	Called when the activity is first created. This is where you should do all of your normal static set up: create views, bind data to lists, etc. This method also provides you with a Bundle containing the activity's previously frozen state, if there was one. Always followed by onStart().	No	onStart()
onRestart()	Called after your activity has been stopped, prior to it being started again. Always followed by onStart()	No	onStart()
onStart()	Called when the activity is becoming visible to the user. Followed by onResume() if the activity comes to the foreground, or onStop() if it becomes hidden.	No	onResume() or onStop()
onResume()	Called when the activity will start interacting with the user. At this point your activity is at the top of the activity stack, with user input going to it. Always followed by onPause().	No	onPause()
onPause()	Called when the system is about to start resuming a previous activity. This is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, etc. Implementations of this method must be very quick because the next activity will not be resumed until this method returns. Followed by either onResume() if the activity returns back to the front, or onStop() if it becomes invisible to the user.	Pre-HONEYCOMB	onResume() or onStop()
onStop()	Called when the activity is no longer visible to the user, because another activity has been resumed and is covering this one. This may happen either because a new activity is being started, an existing one is being brought in front of this one, or this one is being destroyed. Followed by either onRestart() if this activity is coming back to interact with the user, or onDestroy() if this activity is going away.	Yes	onRestart() or onDestroy()

Method	Description	Killable?	Next
onDestroy()	The final call you receive before your activity is destroyed. This can happen either because the activity is finishing (someone called finish() on it, or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the isFinishing() method.	Yes	*nothing*

Each application is compiled into an Android package with the APK file extension which is basically a zip archive. This package contains the compiled code, resources and additional data. This single file is considered an application ready to be installed on a device.

Some of the files that are included are AndroidManifest.xml described classes.dex which contain the classes compiled in Dalvik Executable DEX format understandable by DVM to run the application.

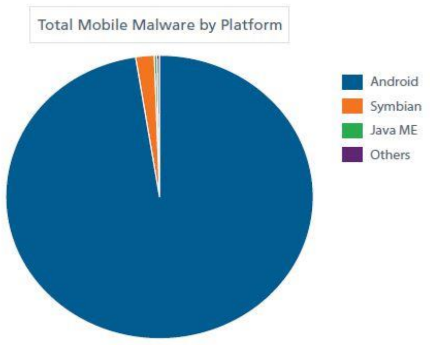
DVM is a register based virtual machine that executes the applications on the platform by interpreting the DEX file containing the compiled classes. The generated Java class files are transformed into the DEX file, however this does not contain Java bytecode, but an alternative instruction set used by the DVM. The Dalvik bytecode assigns for example local variables to any of the available register and the opcodes manipulate directly the registers instead of accessing elements on the program stack.

2.3 Android Malwares

Malicious software is referred to as Malware, classified by its nature as either computer virus, Trojan horse, worm, backdoor or rootkit. the most common Malware types [40] are :

- **Virus:** Code that that inserts itself into another program and replicates, that is, copies itself and infects other computers. Nowadays often used as a generic term that also includes worms and trojans horses.
- **Worm:** Self-replicating Malware which copies itself to other nodes in a network without user interaction using vulnerabilities. Worms do not attach themselves to an application like a virus do.
- **Trojan horse:** Malicious program which masquerades itself as being an application. Unlike viruses and worms, it does not replicate itself.
- **Rootkit:** Software that enables continued privileged access to a computer while actively hiding its malicious activity from administrators by modifying the operating system functionality.
- **Backdoor:** Specialized trojan horse that masquerades itself as an installed program to enable remote access to a system and bypassing normal authentication. Additionally, backdoors attempts to remain undetected.
- **Spyware:** Software that reveals private information about the user or computer system to eavesdroppers.
- **Bot:** Piece of Malware that allows the bot master, i.e. the author to remotely the infected system. A group of infected systems that are controlled are denoted as botnets, instructed by the bot master to perform various malicious activity such as distributed denial of services, stealing private information and sending spam.

The number of Android devices on the market is increasing and so is the number of users. This attracts Malware authors to target Android devices with the intentions of economical profit, stealing private etc... By reading the MacAfee threats report [31] of the first quarter of 2013 we found that Android Malwares have the maximum number in compare with others platform as shown in Figure 2-3, so the total number of Malware samples in MacAfee database is increasing also as shown in Figure 2-4 and Figure 2-5



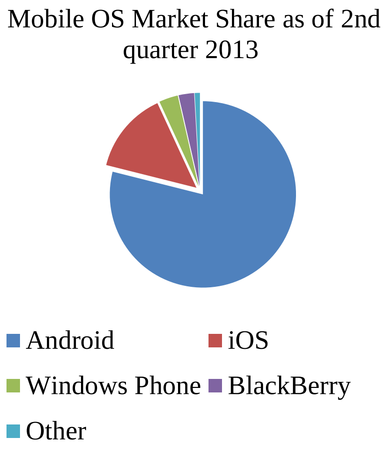


Figure 2-3: Total mobile Malware by platform [31] [18]

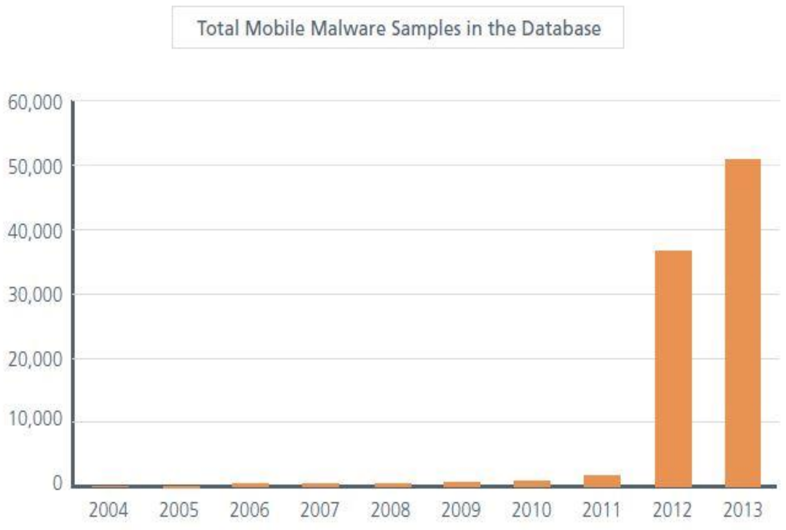


Figure 2-4: Total Mobile Malware Sample in Database [31]

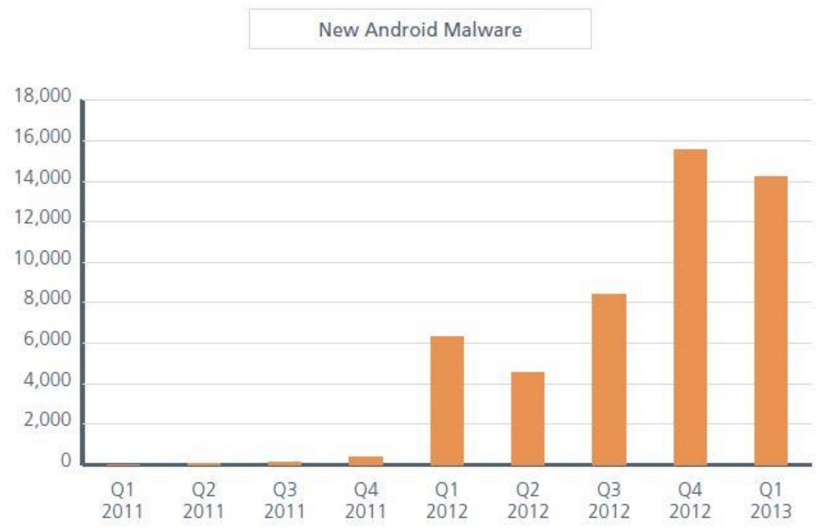


Figure 2-5: New Android Malwares [31]

This ongoing threat emerges from the design of the Android system and Google's policy on releasing applications. The design to isolate applications from each other implies that an application cannot steal or tamper with data belonging to another application. However, using the permissions, an application can be granted access to information from other device subsystems, for example, GPS system and database information such as SMS data and contact entries. So it is still possible for a Malware application to operate within the isolation and still conduct many different categories of attacks and violations, including resource and data loss attacks. The Android permission policy may seem robust but the problem is that this approach relies on the user making the decision whether the combination of permissions used by an application is safe or not. Many users may not have the necessary technical knowledge to make such decisions, and sometimes the users are simply lazy to conduct such inspections when downloading applications.

Android Malware can be distributed easily, compared to iOS applications where a rigorous vetting is conducted. Additionally, Android applications can be released anywhere on the web.

2.4 Android Malware Defense Mechanism

There are two types of code analysis that can be used to detect Malwares, Static Code Analysis and Dynamic Code Analysis, The difference between these two types is that static program analysis is the analysis of software that is performed without actually executing programs while analysis performed on executing programs is known as dynamic analysis, For Android Malware detection there exist many tools for both static and dynamics, below is some of those tools:

- **Static Analysis tools** like IDA Pro, Mobile Sandbox, APKInspecto, and Androguard which is widely used.

  They use different tools to extract features and analyze byte code, this type of analysis will be used for this thesis. More description will be discussed later in chapter 3.
- **Dynamic analysis tools** like SandBox , Droidbox, and AndroidAuditTools used after program execution, In fact dynamic analysis tools are used to analysis code, results from those tools are used by Machine learning to classify application behavior.

Actually there are a lot of Machine Learning tools that can be used for clustering large data set of Android applications features and classify Malware application behavior.

2.5 Related Work

One of the most important tools that can be used for application classification and Malware detection is Machine Learning ML techniques on static features that are extracted from Android's application files, those features are extracted from Android's Java byte-code i.e., .dex files . In [7] authors use this technique and their evaluation focused on classifying two types of Android applications: tools and games, They performed an evaluation using a collection comprising 2,850 games and tools. The results show that the combination of Boosted Bayesian Networks and the top 800 features selected using Information Gain yield will increase the accuracy level of classification.

As it was mentioned before there are two types of Malware code analysis , one of them is dynamic behavior analysis for Android Malware inspection which was presented In [41]. The system consists of a log collector in the Linux layer and a log analysis application. The log collector records all system calls and filters events with the target application. The log analyzer matches activities with signatures described by regular expressions to detect a malicious activity. Signatures of information leakage are automatically generated using the Smartphone IDs, e.g., phone number, SIM serial number, and Gmail accounts. They implement a prototype system and evaluate 230 applications in total. The result shows that the system can effectively detect malicious behaviors of the unknown applications.

In [42] , automatic Malware detection mechanism for the Android platform based on the results from sandbox tool is proposed. They extracted network spatial features of Android apps and used independent component analysis to determine the resolution behavior of Android Malware. The proposed mechanism can identify Android Malware automatically. A public Android Malware app dataset and popular benign apps collected from the Android

Market are used for evaluating the effectiveness of the proposed approach in terms of its grouping ability and effectiveness in identifying Android Malware.

To identify possible information leakage, LeakMiner [35] applies a static taint analysis to apps within Android market. The approach introduces three steps in identifying possible leakages: first, apk files of Android apps are transformed to Java bytecode so that the following analysis can directly work on Java bytecode. Besides, application metadata are extracted from the manifest file of Android app. Then, LeakMiner identifies sensitive information according to the extracted metadata. Finally, taint information is propagated through call graph to identify possible leakage paths. By introducing multiple entry point call graph, They can cover all the code of Android app. They choose a set of 1750 apps to evaluate the accuracy of LeakMiner. LeakMiner can identify 145 real leakages in this app set.

A feature-based mechanism to provide a static analyst paradigm for detecting the Android Malware proposed in [46]. The mechanism considers the static information including permissions, deployment of components, Intent messages passing and API calls for characterizing the Android applications behavior. In order to recognize different intentions of Android Malware, different kinds of clustering algorithms can be applied to enhance the Malware modeling capability. Furthermore, They leverage the proposed mechanism and develop a system, called DroidMat which extracts the information e.g., requested permissions, Intent messages passing, etc from each application's manifest file. In addition, it traces API calls for each component since API calls in different components may imply different intentions. Then, it applies K-means algorithm that enhances the Malware modeling capability. Finally, it uses kNN algorithm to classify the applications benign or malicious.

In [14] In the first part the paper presented several algorithms to compare applications to identify their similarities or differences. To do that they applied an original selected compressor (Snappy) obtain a real usable tool to improve the time of comparison with good results. With this similarity distance, they created a tool to determine whether a version of application has potentially been pirated. Next they applied this technique to design a tool to measure the efficiency of an obfuscator, and they demonstrated that there is a lack of proven tools in this domain. In the final part they described a new algorithm to find and visualize dissimilarities between versions of an application. The tools and the framework are open source and can be downloaded on the website.

In [1] authors explain how to apply clustering techniques in Malware detection of Android applications. they also use machine-learning techniques in auto detection of Malware applications in the Android market. Their evaluation is given by clustering two categories of Android applications: business, and tools. They have extracted 18,174 Android's application files in their evaluation using clustering. They extract the features of the applications from applications' XML-files which contains permissions requested by applications. The results gives a positive indication of using unsupervised machine learning techniques in Malware detection in mobile applications using a combination of the application information and xml AndroidManifest files .

This research is based on the idea of enhancing the similarity measurement to find Malwares by using SimHash algorithm which will be discussed later in chapter 3 and 4. This idea is close to one presented in [32] but they use SimHash to enhance detection of clones codes in large system which lead to unresolved bug or maintenance related problems by increasing the risk of update anomalies, they investigate the effectiveness of SimHash, a state of the art fingerprint based data similarity measurement technique for detecting both exact and near miss clones in large scale software systems they took an existing code cloning system and improved the time performance by an order of magnitude using SimHash, and demonstrated its feasibility for use with large systems such as the Linux Kernel. As well, they adapted SimHash to a code cloning framework and demonstrated its viability for the clone detection in large scale systems.

2.6 Summary

Background about Android system, application and programming has been discussed from Android website we talk about Android architecture, components and activity life cycle; This chapter also talks about Malwares in general and Malware in Android applications with more details which has been extracted from McAfee 1st Quarter 2013 report, at the end of this chapter there is a group of related work in the topic of Android Malwares detection and SimHash algorithm .

3 Research Tools

Malwares for Android application are considered as one of the most growing problems, so there must be a new techniques and tools to detect these Malwares. In fact there are many antiviruses' tools in today market to detect Malware using either static or dynamic analysis, In this chapter a scientific view will be presented for algorithms and techniques used for static code analysis .

3.1 Reverse Engineering

Reverse Engineering is a process of analyzing program code or software in order to test it from any vulnerability or any errors. Reverse engineering is the ability to generate the source code from an executable code. This technique is used to examine the functioning of a program or to evade security bugs, etc. Reverse engineering can therefore be stated as a method or process of modifying a program in order to make it behave in a manner that the reverse engineer desires. Joany Boutet has quoted Shwartz, saying, "Whether it's rebuilding a car engine or diagramming a sentence, people can learn about many things simply by taking them apart and putting them back together again. That, in a nutshell, is the concept behind reverse-engineering - breaking something down in order to understand it, build a copy or improve it " [24].

From the beginning of 2009 research scientists began proposes tools for reverse the DalvikBytecode. One of them is "undX" tool which could generate a JAR file from an Android APK file, then convert to JAVA using tools such as JAD and and JD-GUI. The "undX' tool worked well with basic applications; but it posed many problems when dealing with complex Dalvik Bytecode. The Dex2Jar tool originated then. Dex2Jar does similar job to “undX”; but this tool also has some issues while dealing with complex Dalvik Bytecode .

There are many programs which have reversed different applications of Android from byte code to readable code in order to study the vulnerabilities like Androguard tools.

3.2 Androguard

Androguard is mainly a python tool done by VirusTotal project, VirusTotal is a subsidiary of Google, is a free online service that analyzes files and URLs enabling the identification of viruses, worms, trojans and other kinds of malicious content detected by antivirus engines and website scanners. At the same time, it may be used as a means to detect false positives, i.e. innocuous resources detected as malicious by one or more scanners. Virus Total mission is to help in improving the antivirus and security industry and make the internet a safer place through the development of free tools and services [44].

Androguard play mainly with:

- Dex/Odex Dalvik virtual machine , .dex disassemble, Decompilation .
- APK Android application .
- Android's binary xml.
- Android Resources.

Androguard has the following features **:**

- Map and manipulate DEX/APK format into full Python objects.
- Disassemble/Decompilation/Modification of DEX/APK format.
- Decompilation with the first native directly from dalvik byte codes to java source codes dalvik decompiler .

- Access to the static analysis of the code basic blocks, instructions, permissions and create analysis tool.
- Analysis a bunch of Android apps.
- Diffing of Android applications.
- Check if an Android application is present in a database.
- Open source database of Android Malware
- Reverse engineering of applications
- Transform Android's binary xml like AndroidManifest.xml into classic xml.

The most important feature that has been used in thesis is similarity measurement which will be used later to detect Malwares after measuring distance between it and given clustered datasets of Malwares families. Androguard uses Elsim project to measure distance between two text codes of different Android applications, Elsim has an important tool which used to measure similarity, this tool is called Androsim , This tool detects and reports identical methods, similar methods, deleted methods, new methods and skipped methods.

Moreover, a similarity between 0.0 to 100.0 is calculated upon the values of the identical methods and the similar methods. Androguard calculate the final values using text compressor. It is more interesting because an understandable value related to the similarity will be discovered.

Elsim uses Normalized Compression Distance NCD, Which is way of measuring the similarity between two objects, either they are two documents, two letters, two programs…etc. Such a measurement should not be application dependent or arbitrary. A reasonable definition for the similarity between two objects is how difficult it is to transform them into each other. Next Section will talk about measuring distance by compression.

3.3 Similarity Measurement by Compression

There are several concepts to measure similarity of strings or files. One of the most important theoretical fundamentals is the Normalized Compression Distance NCD [19] ,Although there are many known methods, that measure the similarity between two given data pieces with specific content, such as image, text or audio files, most of them are highly specialized, complex and work on high level aspects of the data. The idea behind the compression distance is to measure closeness of any given files or strings without regarded of their specific content or structure taking in mid process of fast and simple algorithms that work on a very low level.

3.4 Information Distance

The concept of Information distance between two strings x and y can be defined as the length of the shortest program p that computes x from y and vice versa. This shortest program is in a fixed programming language. For technical reasons one uses the theoretical notion of Turing machines. Moreover, to express the length of p one uses the notion of Kolmogorov complexity $K(x)$ which is a measure of the computability resources needed to specify the object. So p will be as define as shown below:

$$|p| = \max\{K(x|y), K(y|x)\}.$$

The Kolmogorov complexity can be considered as an idealized measurement of the informational quantity given in a string or file. Based on this idea the informational distance of two strings could also be measured by consulting the concept. In doing so, we are talking about Turing machines that are given the information of one string to resemble the other one. The less different the two strings are, the less complex this task should be and therefore the less the size of the smallest Turing machine, that does the job. To formalize this thought, we define the Normalized.

Normalized Information Distance of two strings $\mathrm{x}, \mathrm{y} \in \sum *$ as

$$NID(x,y) = \frac{\max\{K(x|y), K(y|x)\}}{\max\{K(x), K(y)\}}$$

where K(x|y) is the conditional Kolmogorov complexity, i.e. the size of a smallest Turing machine, that, given y as input, generates x on its output tape and then halts. The normalization term $\max\{\mathrm{K(x)}, \mathrm{K(y)}\}$ is included to generalize the approach on parameters of variable size and informational quantity. the NID minorizes all normalized functions in a wide class of relevant distance functions, meaning, that if two strings are close according to an aspect measured by one of these functions, they are also close according to the NID.

3.4.1 Compression Distance

Even if we want to accept the NID as a generalized measurement for informational distance, there remains the problem of non-computability of K(x). One obvious real world approximation to the NID is to see the Turing machines as a maximal possible compression of the string x. Therefore the size of the output C(x) which is the binary length of the file x compressed with compressor C for example "gzip", "bzip2", "PPMZ", produces on input x could be taken to approximate K(x) in the formula. So $\max\{\mathrm{K(x|y)}, \mathrm{K(y|x)}\}$ could be approximated by $\mathrm{C(xy)} - \min\{\mathrm{C(x)}, \mathrm{C(y)}\}$, where xy is the concatenation of x and y. This way it can be seen how good the compression algorithm can use the information given in one string to better compress the other one and vice versa. By subtraction of $\min\{\mathrm{C(x)}, \mathrm{C(y)}\}$ the term approximates the remaining compressed size of the more complex string if the information in the other one was used. Therefore the term should be smaller, the more similar the two strings are. Considered together, we define the Normalized Compression Distance as

$$NCD(x, y) = \frac{C(xy) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}}$$

Obviously there is no way to prove that this function approximates what would be seen as an intuitive distance of file contents, like similarity of pictures or tunes, i.e. giving an absolute and adequate universal and parameter free similarity metric. But the NCD summarize distance functions up to a dedicated term, according to the quality of the approximation of K(x|y) by the chosen compressor.

3.5 SimHash Algorithm

Measuring similarity between two strings has many algorithms one of them is SimHash algorithm which has been created by Moses Charikar [12] from Google. It can compare easily between two strings, and by the way two datasets of any type, quickly and effectively.

SimHash use a fingerprint system, instead of comparing the texts directly, it will compare their fingerprints, which is really more effective and usable in reality.

This chapter will offer a complete description of this algorithm which has been used next to detect Malware in Android applications

3.5.1 The Similarity Problem for SimHash

The similarity computing is not easy because recognize similar element that are not equals is complex with a computer. To compare two things, the human brain create a list of criteria for each item [17], SimHash will do the same, it will compare two texts using each word as a crtieria that describe the text. In maths, we can sum up it as:

$$sim(A, B) = \frac{A \cap B}{A \cup B}$$

This is the Jaccard index. However, this index is not effective at all as it requires checking for each element in A if there is an equal element in B to check for each word in the first text if there is an equal word in second one.

SimHash is the process in which fingerprints will be created to compare the texts. Therefore, $(A \cap B)$ will be replaced by the comparison between their binary fingerprints, which is much more effective. To generate those fingerprints, Moses Charikar's algorithm steps are:

```
1- Define a fingerprint size (for instance 32 bits)
2- Create an array V[]  filled with this size of zeros
3- For each element in the dataset, create a unique Hash with md5, sha1
   of any other Hash algorithm that give same-sized results
4- For each Hash, for each bit i in this Hash

      If the bit is 1, we add 1 to V[i]

      If the bit is 0, take 1 to V[i]

5- For each i

      If V[i] >0, i = 1

      If V[i] < =0, i = 0
```

After that, fingerprint will be created which characterizes the text. Now, to compare two fingerprints, you just have to compare two binary numbers: XOR logical will be used and count the 1 in the result Hamming Distance.

$$10101011100010001010000101111100$$

$$\text{XOR}\ 10101011100010011110000101111110$$

$$=\ 00000000000000010100000000000010$$

We have 3 ones for 32 characters, so we have 3 differences on 32 elements : the estimation of the difference is $3/32 = 0{,}09375$ so the estimation of the similarity is $1 - 3/32 = 0{,}90625$ a bit more than 90%.

3.5.2 SimHash Text Example

If we want to measure the similarity between two sentences bellow

1- "the cat in the tree is white"
2- "the man in the suit is happy"

First we extract text features and generate simple md5 32 Hash codes for each feature;

Note: In this example sentence words are considered as features. In thesis experiment each 2 chars from sentence are considered as feature.

Features	**Hash MD5 first 32bit**
the	a53794d2
cat	54b8617e
in	ba8d2b94
the	a53794d2
tree	c0af77cf
is	a2a551a6
white	d508fe45
man	39c63ddb
suit	9cf5af27
happy	56ab24c1

As in algorithm each word will be converted to binary 32bit binary Hash code then all Hash codes above will grouped in one Hash called SimHash code (for each bit if the number of 1 greater than number of zeros then output is one else output will be zero)

Word	**Binary Hash**
the	10100101001101111001010011010010
cat	01010100101110000110000101111110
in	10111010100011010010101110010100
tree	10100101001101111001010011010010
is	10100010101001010101000110100110
white	1010101000010001111111001000101
SimHash Code	**10100000001101010001000010010010**

Word	**Binary Hash**
the	10100101001101111001010011010010
man	00111001110001100011110111011011
in	10111010100011010010101110010100
suit	10011100111101011010111100100111
is	10100010101001010101000110100110
happy	01010110101010110010010011000001
SimHash Code	**10110000101001010010010110000010**

Then we find hamming distance which is number of one taken from the result of binary XOR between SimHash code for each sentence which is equal to **1000010010000000110101000010000=8 ones**

So the distance will be 8/32 = 25% so the similarity =1-.25 =0.75 ~75%

3.6 Summary

This chapter describes in theoretical view of the most important used tools in this thesis; First a complete description about Androguard project with its usage and features has been provided, then a scientifically description about Information distance concept has been discussed and additional information about Androguard information distance algorithms is explained also which is called compression distance .Finally this chapter talks about SimHash algorithm, its concept , usage and features with an example for more clear view .

4 Methodology Evaluation and Analysis.

In this chapter a methodology, experiments and thesis proposed algorithm will be discussed. In Section 4.1 thesis new method of measuring similarity between two Android application and detect Malwares will be discussed and it will be compared with Androguard tool, Section 4.2 discusses the first experiment with its input attributes and its results, Section 4.3 is the second experiment and its evaluation, . Finally section 4.5 is Summary for chapter.

4.1 Malware detection by SimHash algorithm

In this thesis we propose a new method which measures similarity between two Android applications to detect Malware application by using output of Androguard as reverse engineering tool and SimHash algorithm.

As we mentioned before Androguard is an open source project done by VirusTotal group this project has many functions for Android application static code like reverse engineering, Malware detection and similarity measurement tools.

SimHash algorithm [12] has been used to find the distance between two stings or two file by converting them to Hash codes of 32 or 64 bit, the idea of SimHashing two string is to convert similar strings to similar Hashes and then compare between two Hashes which makes the process of comparison with low time and low resources consumption.

In our proposed method we make a static code analysis so the input will be .apk file which is the setup file of Android application, the first step is to use Androguard "androlyze.py" and "get_package()" tool to extract and analyze .apk file, Androlyze program has many functions it could disassemble an Android application , it has "get_dex()" method which return a content of dex code , compiled code by Dalvik machine for Android application. the return of this function has been used as the string input for SimHash algorithm. “get_package()” which returns the content of manifest.xml file the file which control the permission of Android programs and has a summarized information of all program activities will also concatenating to string as an input.

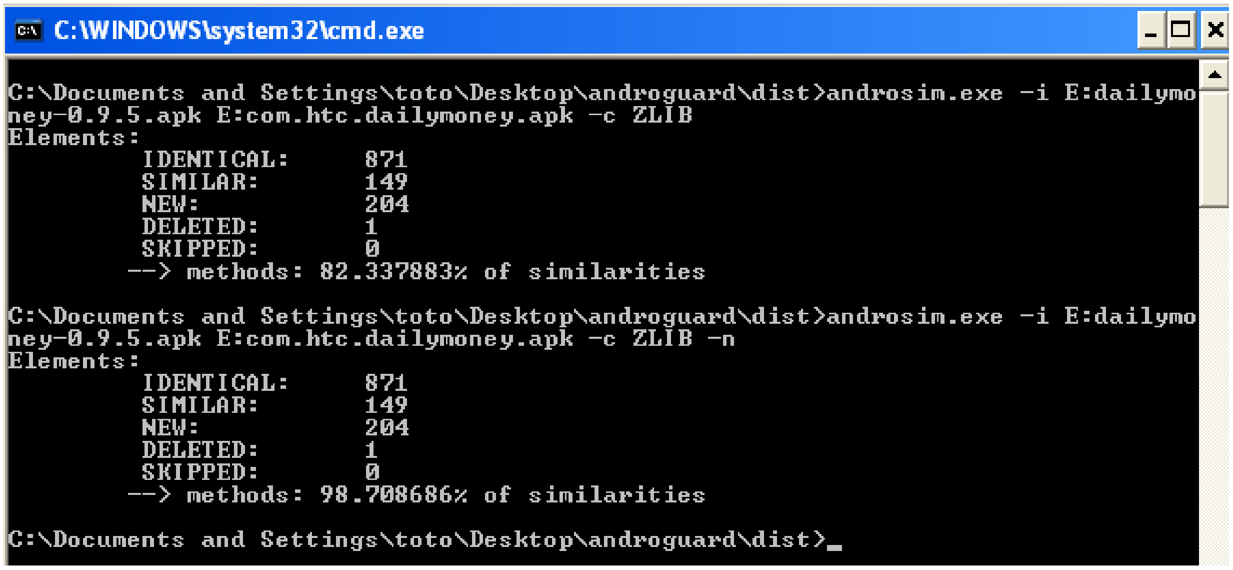
In step two we use SimHash the generated .Dex , .XML from .apk files, the result of hamming distance of those two Hashes will be the distance between applications.

Finally step three will use step one and two but with an existing database of Malware application SimHash signatures, if the results are very closed which means it is more than the threshold, then it will consider the input as Malware. If not it will consider this application as legal and not worm or Malware application.

So proposed algorithm can be summarized as follow

```
1- Input .apk file
2- Use  Androlyze. get_dex() and get_package()  method to get content
   of .DEX compiled code and .XML manifest file
3- SimHash .DEX  and .XML for inputs
4- Measure similarity with exists Malware  database signatures by using
   hamming distance
5- If the output > threshold… input is Malware.
```

In fact Androguard has tools to measure similarity between Android applications like Elsim or androsim program, this tool detects and reports the identical method, the similar methods, the deleted methods, the new methods, and skipped methods as in Figure 4-1 below. It takes signature of each application and then adds it to database which will be used next to detect similar applications and Malwares.

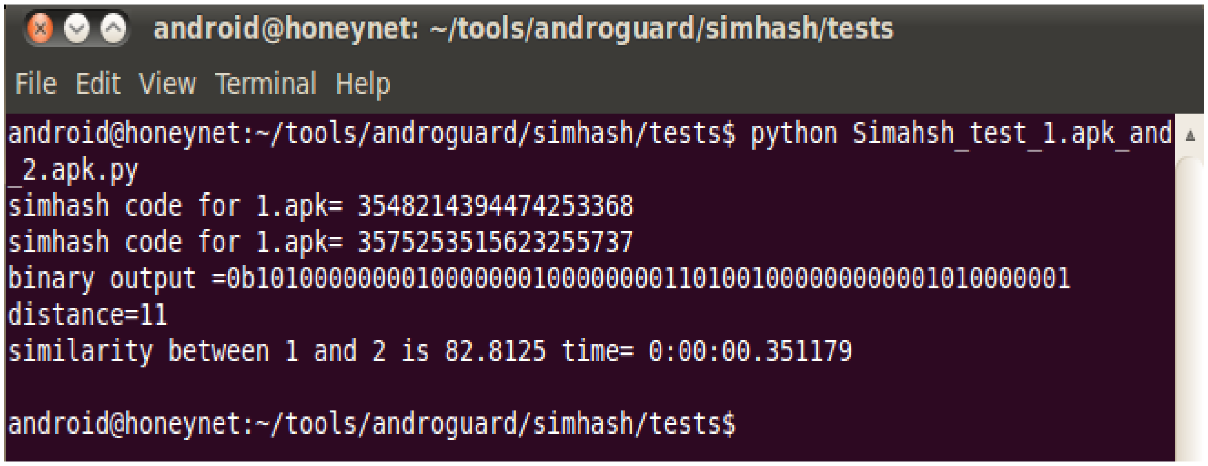


```
C:\Documents and Settings\toto\Desktop\androguard\dist>androsim.exe -i E:dailymo
ney-0.9.5.apk E:com.htc.dailymoney.apk -c ZLIB
Elements:
         IDENTICAL:     871
         SIMILAR:       149
         NEW:           204
         DELETED:       1
         SKIPPED:       0
        --> methods: 82.337883% of similarities

C:\Documents and Settings\toto\Desktop\androguard\dist>androsim.exe -i E:dailymo
ney-0.9.5.apk E:com.htc.dailymoney.apk -c ZLIB -n
Elements:
         IDENTICAL:     871
         SIMILAR:       149
         NEW:           204
         DELETED:       1
         SKIPPED:       0
        --> methods: 98.708686% of similarities

C:\Documents and Settings\toto\Desktop\androguard\dist>_
```

Figure 4-1: Androsim to Find Similarity Between Two .apk Programs

Androguard is an efficient tool and it is used by VirusTotal which consider one of the most important groups that develop new strategies for antiviruses, but it has some disadvantages, it takes a lot of time in RE process while generating application signature, especially for large size applications. In our proposed algorithm we try to evade this disadvantage with results similar to Androguard, So we have used little part of RE which analyze .DEX and .XML part only and we use SimHash to generate signature of this application, it utilize time since the signature will be with the same size and finding distance will be in very easy method by hamming distance.

Our proposed method accuracy will not the same of Androguard tools, since it used to analyze application completely in very complicated process which depends on may attributes, but we depend mainly on .DEX compiled code which contains the Malware behavior and XML, this will make our results near or similar to the results from Androguard, but we save a lot of time and resources as it will be seen next chapter. Figure 4-2 below shows an example of output of thesis new method.


android@honeynet: ~/tools/androguard/simhash/tests
File Edit View Terminal Help

```
android@honeynet:~/tools/androguard/simhash/tests$ python Simahsh_test_1.apk_and
_2.apk.py
simhash code for 1.apk= 3548214394474253368
simhash code for 1.apk= 3575253515623255737
binary output =0b1010000000010000000100000000110100100000000000001010000001
distance=11
similarity between 1 and 2 is 82.8125 time= 0:00:00.351179

android@honeynet:~/tools/androguard/simhash/tests$
```

Figure 4-2:SimHash To Find Similarity Between Two .apk Programs

4.2 Experiment 1: Comparison with Androguard.

In this Section an experiment to compare thesis modified method with an existing widely used tool Androguard will be explained to find and discuss advantages and disadvantages, the experiments result will be presented and analyzed as shown below.

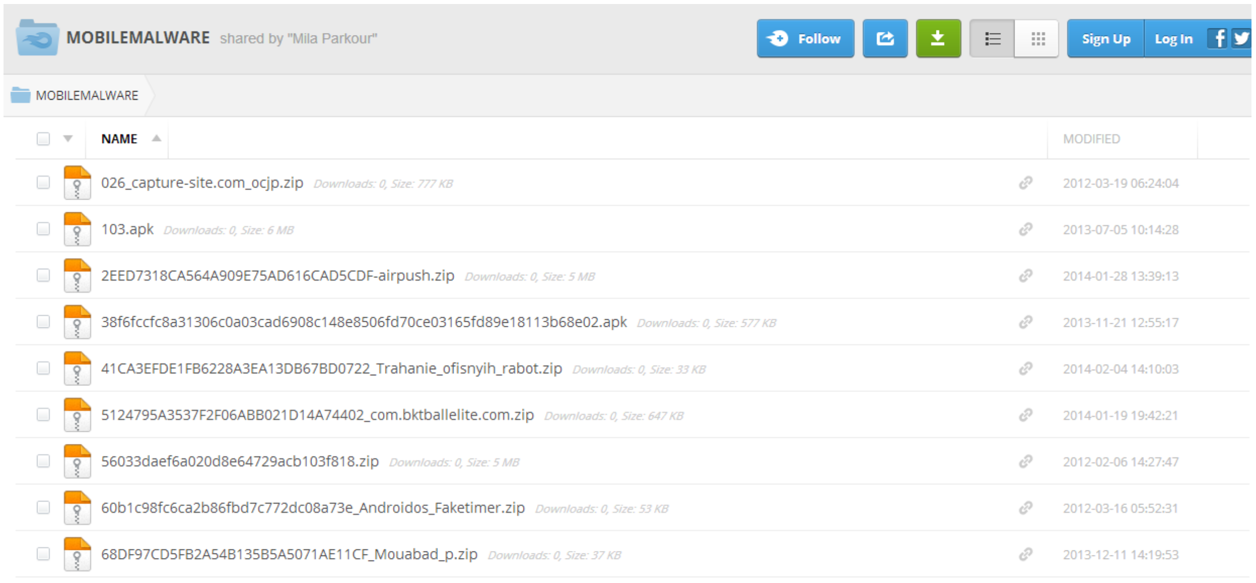
4.2.1 Experiment Environment:

Androguard library has been installed on Ubuntu Linux based operating system which has been setup as virtual machine run by oracle virtual box on Dell OptiPlex 9010 brand name pc (CPU: Intel core i5-3470 @3.2GHz 4 CPUs,RAM:8G, OS: Windows 7 64bit) ,the virtual machine setting was as follow:(Processor: 2CPU execution cap:100% and Ram:4G,OS:Ubunto4.3) , Also our new program has been written on the same virtual machine with python programming language and it has used some of libraries from Androguard tools as shown in appendix.

4.2.2 Experiment Inputs and Dataset

In order to measure and evaluate proposed method, different experiment should be occurred on a training data of Android application, Smartphone malicious applications can be collected from several open source sites such as Contagion, Offensive and VXHeavens. In this thesis training data has been collected from Contagiodump which provides smartphone Malware that infects various smartphone platforms such as Android, iPhone, BlackBerry and Windows mobile, Contagio mobile mini-dump is a part of contagiodump.blogspot.com. Contagio mobile mini-dump offers an upload dropbox for you to share your mobile Malware samples.

A complete dataset of different Malware Android applications (130 different Malware) has been downloaded from Contagiodump, All experiments have been done on those applications as will be shown in next Sections. Those applications are different in functions, types, source and type some are business applications, games ,social and….etc., All applications have been downloaded from http://www.mediafire.com/?78npy8h7h0g9y link as in Figure 4-3 shown below.



NAME		MODIFIED
026_capture-site.com_ocjp.zip	Downloads: 0, Size: 777 KB	2012-03-19 06:24:04
103.apk	Downloads: 0, Size: 6 MB	2013-07-05 10:14:28
2EED7318CA564A909E75AD616CAD5CDF-airpush.zip	Downloads: 0, Size: 5 MB	2014-01-28 13:39:13
38f6fccfc8a31306c0a03cad6908c148e8506fd70ce03165fd89e18113b68e02.apk	Downloads: 0, Size: 577 KB	2013-11-21 12:55:17
41CA3EFDE1FB6228A3EA13DB67BD0722_Trahanie_ofisnyih_rabot.zip	Downloads: 0, Size: 33 KB	2014-02-04 14:10:03
5124795A3537F2F06ABB021D14A74402_com.bktballelite.com.zip	Downloads: 0, Size: 647 KB	2014-01-19 19:42:21
56033daef6a020d8e64729acb103f818.zip	Downloads: 0, Size: 5 MB	2012-02-06 14:27:47
60b1c98fc6ca2b86fbd7c772dc08a73e_Androidos_Faketimer.zip	Downloads: 0, Size: 53 KB	2012-03-16 05:52:31
68DF97CD5FB2A54B135B5A5071AE11CF_Mouabad_p.zip	Downloads: 0, Size: 37 KB	2013-12-11 14:19:53

Figure 4-3 Contagiodump Dataset

4.2.3 Comparison between Androguard and SimHash.

To compare between Androguard and Modified Androguard with SimHash as a tool for similarity measurement, we have to test two tools on different similarity processes,. First; We have made pairing between all applications from downloaded dataset, about 130 applications produce a number of 8257 measure similarity process measured by Androguard and SimHash as shown in Table 4.1 below

Table 4-1: Similarity Measurement, Androguard and SimHash

Similarity Measurement Method	# of comparison	Total time(hours)	Average time for comparison (sec)
SimHash	8257	11	5
Androguard	8257	39	13

As shown from Table there are a big different in computation time for two methods our proposed algorithm is faster than Androguard which takes about 3 time more than SimHash tool, this was because of many reasons , the most important one is that complexity of Androguard which make a lot of complicated process to find signature of Malware or Android application, it uses Elsim tool which was discussed before, this tool analyze the reversed code to find identical method, similar methods, deleted methods, new methods and t skipped methods, this will take a lot of time in addition it uses NCD which was discussed also before , to calculate the similarity of text, NCD takes a lot of time with $O(n^2)$ [19] complexity , In other hand our proposed algorithm do a little simple process after reverse engineering process it easily SimHash the reversed code .DEX and .XML by using SimHash algorithm then it use Hamming distance to calculate the similarity this will of course save time with $O(n\log(n))$ [12] complexity and memory and will generate signature in very easy way.

After applying two algorithms we collect a very huge data of comparisons which will help us to evaluate our modified method and compare it with Androguard. After generating data from SimHash we found that all data are centralized between (75%-100%) so it is preferred to normalize data to be between (0%-100%)   in order to compare it with Androguard. Figure 4-4 below so the comparisons process of random applications. As it shown from Figures , results are very close overall processes sometimes there are different but in small similarities, but in general similarity distance are identical for values more the 70%, this indicates that proposed method has a very good result near to Androguard but in small time,   as shown from time Figures below some processes takes about two minutes by Androguard and takes little than one minutes by SimHash with the same similarity distance in both ways, this also shows that proposed method are very efficient for time and resources saving .

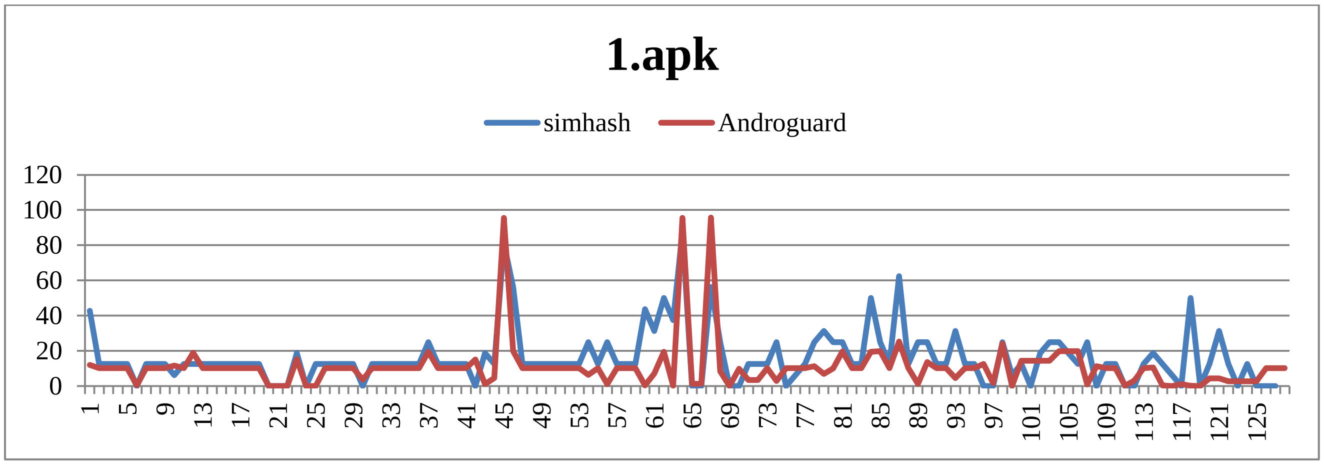


Figure 4-4: Similarity Measurements for 1.apk And Dataset by SimHash and Androguard

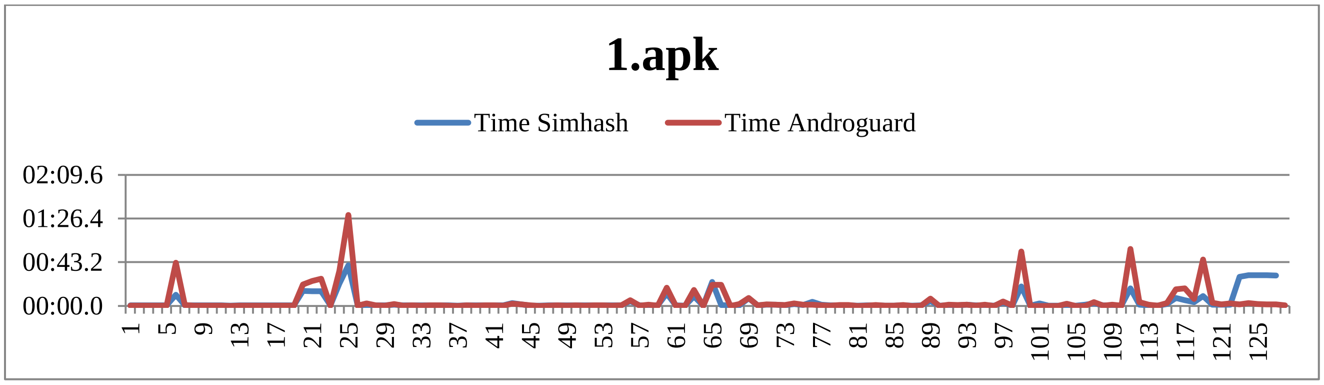


Figure 4-5: Time for Similarity Measurements for 1.apk and Dataset by SimHash and Androguard

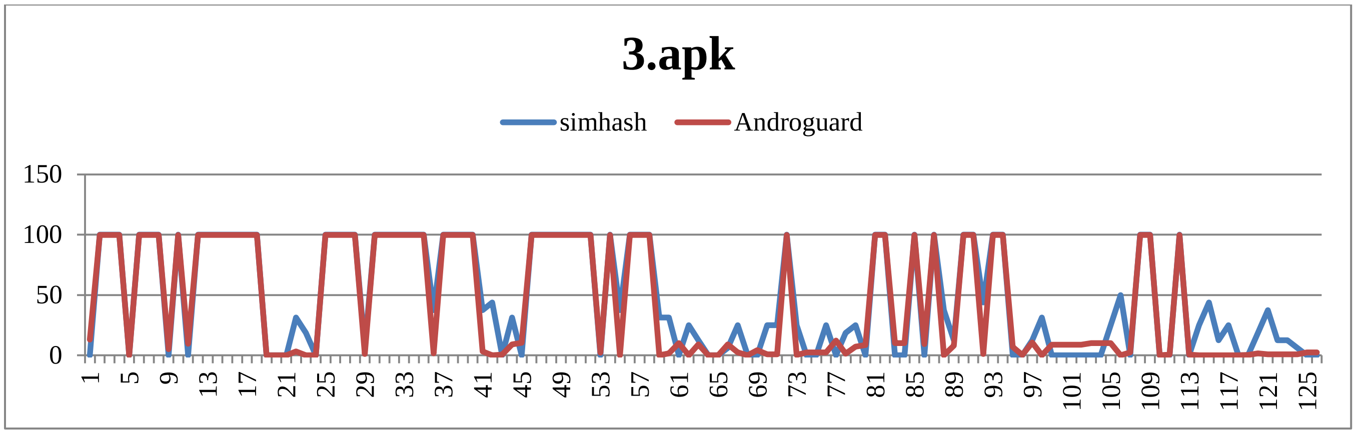


Figure 4-6: Similarity Measurements for 3.apk and Datset by SimHash and Androguard

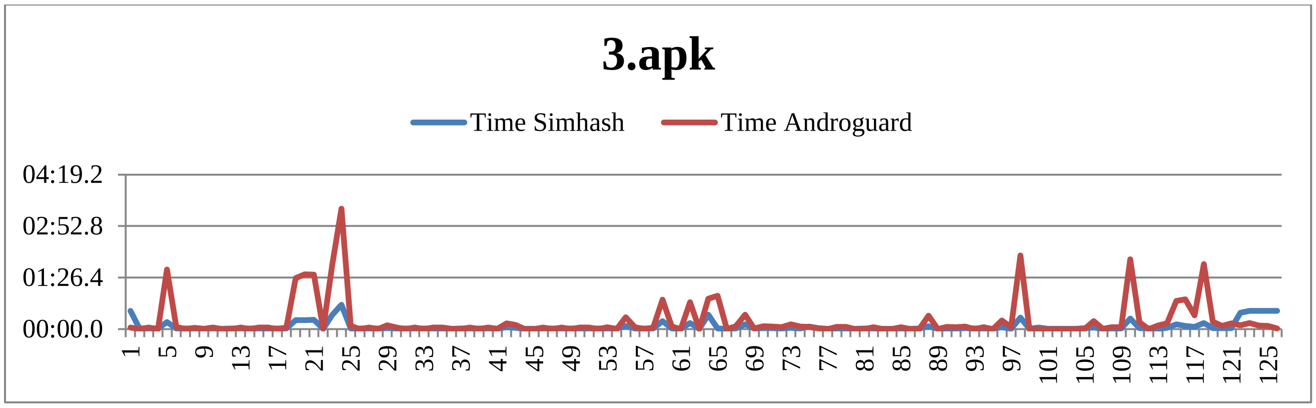


Figure 4-7: Time for Similarity Measurements for 3.apk and Dataset by SimHash and Androguard

To study the time behavior of proposed method and to ensure its feasibility, A time distribution model has been studied for the same experiment results, We have use SPSS PASW statistics Release 18 in order to generate histogram model for collected data, In statistics, a histogram is a graphical representation of the distribution of data. It is an estimate of the probability distribution of a continuous variable and was first introduced in [38] A histogram is a representation of tabulated frequencies, shown as adjacent rectangles, erected over discrete intervals (bins), with an area proportional to the frequency of the observations in the interval. The height of a rectangle is also equal to the frequency density of the interval, i.e., the frequency divided by the width of the interval. The total area of the histogram is equal to the number of data. The histogram output as in Figures below show that experiment takes similar time distribution behavior for both method Androguard and SimHash, they act as exponential distribution not as normal distribution this indicate that overall measurements process are concentrate at period between 1-10 sec , for both Androguard and SimHash, but it is noticed that SimHash has more distribution for that period between 1 and 30 sec that major process are accrued at this time while Androguard has more distribution at long time periods, this ensure the feasibility of our proposed method for time saving.

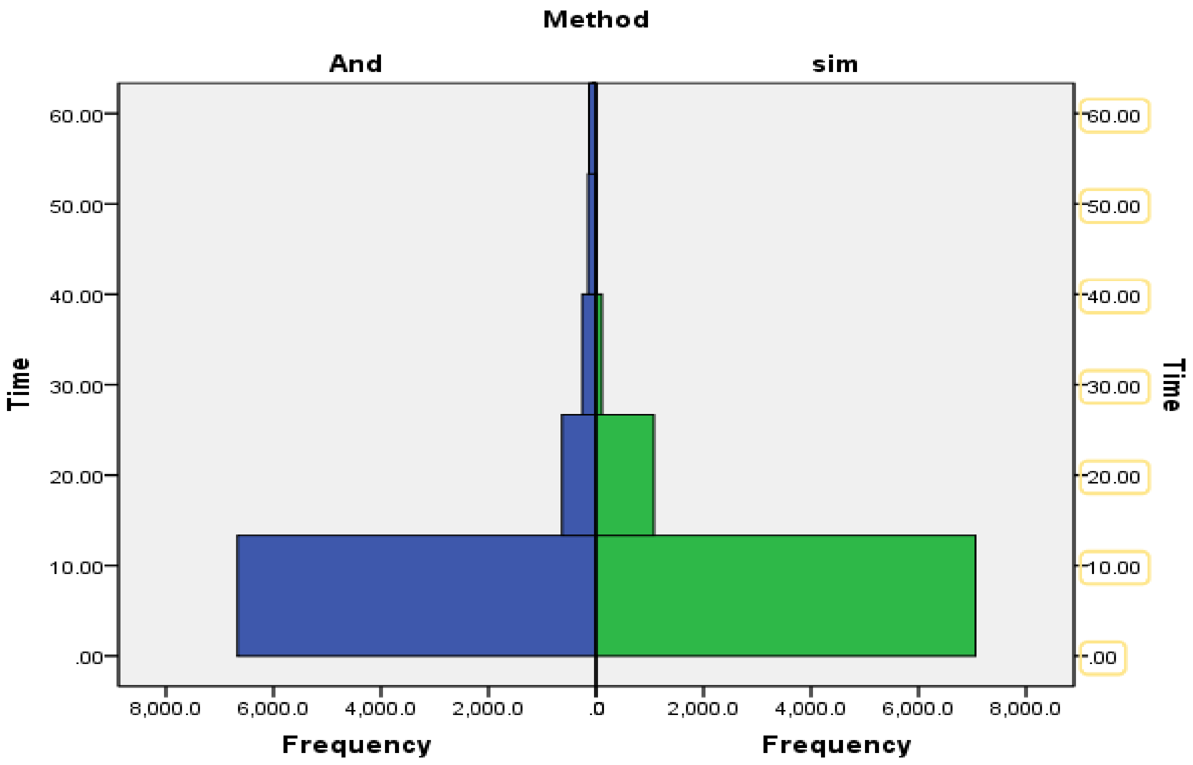


Figure 4-8: Histogram for SimHash and Androguard (1)

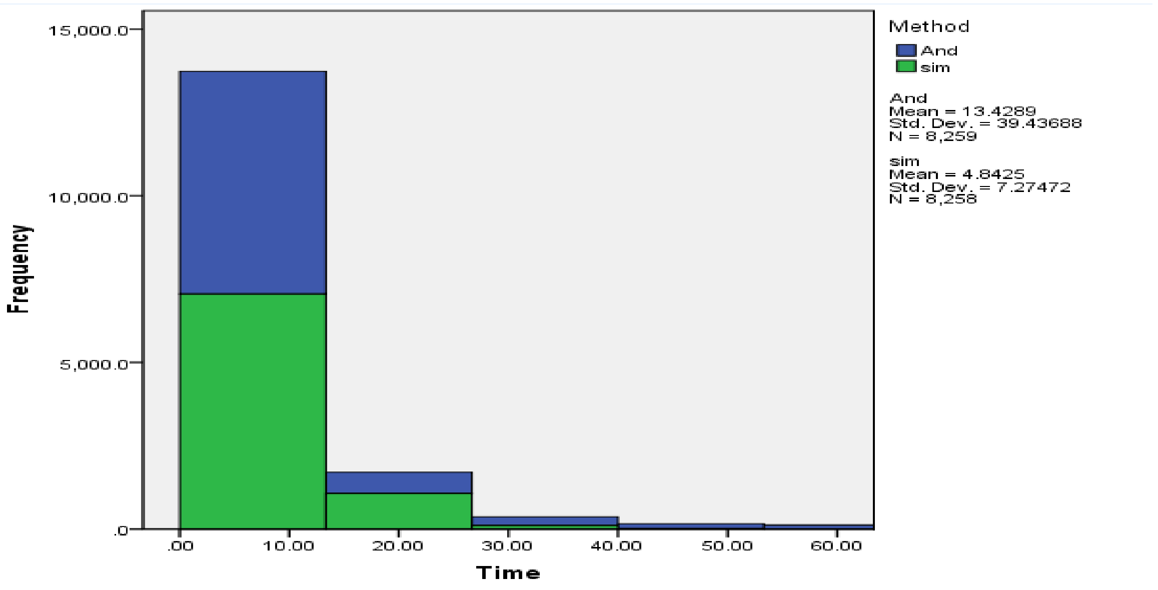


Figure 4-9: Histogram for SimHash and Androguard (2)

4.3 Experiment 2: Malware Detection.

The first experiment shows that thesis proposed algorithm saves a lot of memory and time resources during the process of similarity measurement, in this experiment it is important to show the ability of detecting a Malware by new method, so the input of this experiment will be a real Malware injected application and the same application but without injection, the goal is to detect the injected one by our algorithm.

First of all we use .apk downloaded dataset to generate a database of SimHash signatures, then we insert a test Malware in order to find if it will be detected and if it will be similar to another Malware.

The same process sequence will be used for Androguard which, and the result of two algorithms will be compared.

Table 4-2: Dataset Information

Dataset source link	http://contagiominidump.blogspot.com/
# of applications	130
Types of application	Games, social, education…etc.
Shared by	Mila blog

We used angry birds cheat application as a test application, first we install it from Google play as shown in image below, install this application form Google play insure that it it has been inspected by Google static analysis tool,

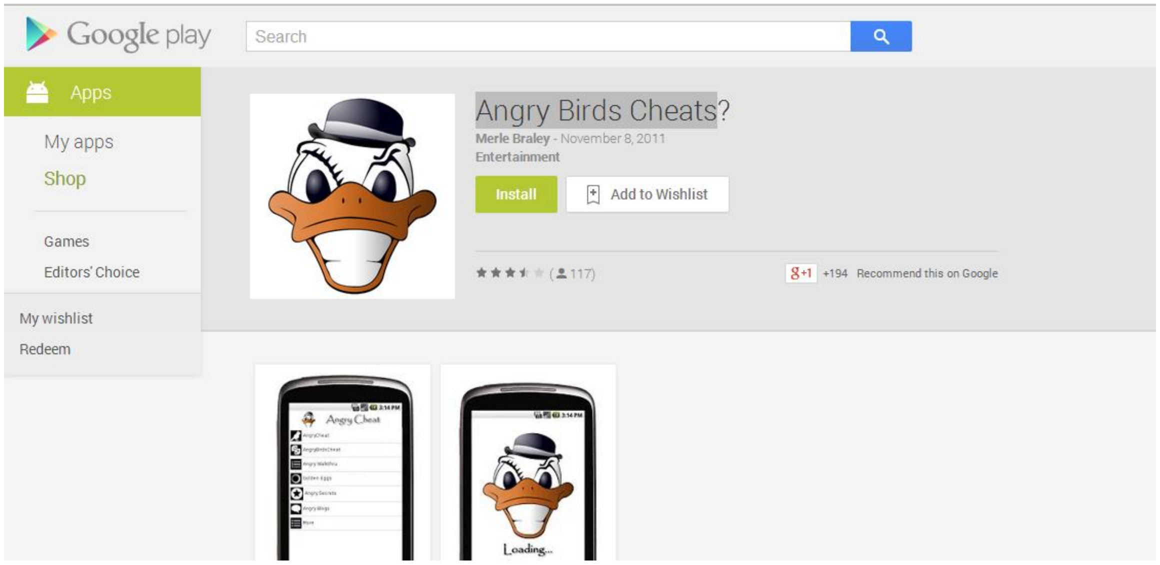


Figure 4-10: Tested Application from Google Play Store

In other hand we download the same application but with Malware from contagion dump blog, but it is not from dataset we have download before, We enter the two .apk applications in to SimHash and Androguard to measure it similarity with data source elements, first the

Table 4-3: Tested Application Information

Application name	Angry Bird cheats
Done By	Merle Braley
Initiate date	November 8, 2011
Type	Entertainment
Descriptions : You'll Learn Secrets that Most Players will Never Know. When you install this app now, you can watch the Official Angry Birds Walk-throughs, PLUS you'll learn about Golden Eggs, Angry Birds Cheats and Many More Secrets. This app is for anyone who love to play Angry Birds!	

injected application has been detected by two algorithms since it takes score more than 80% two times as shown in Figure 4-11 below

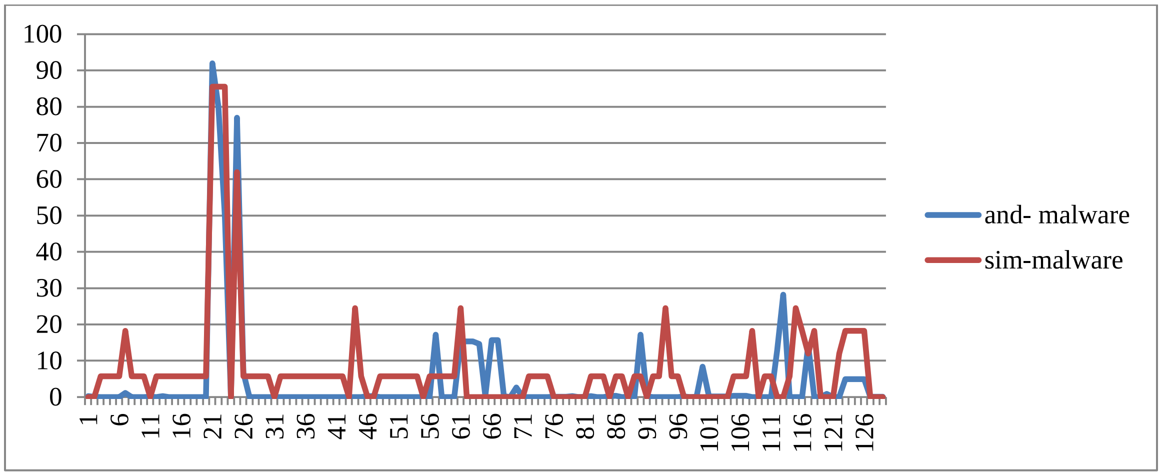


Figure 4-11: Similarity Measurement for Tested Application with Malware

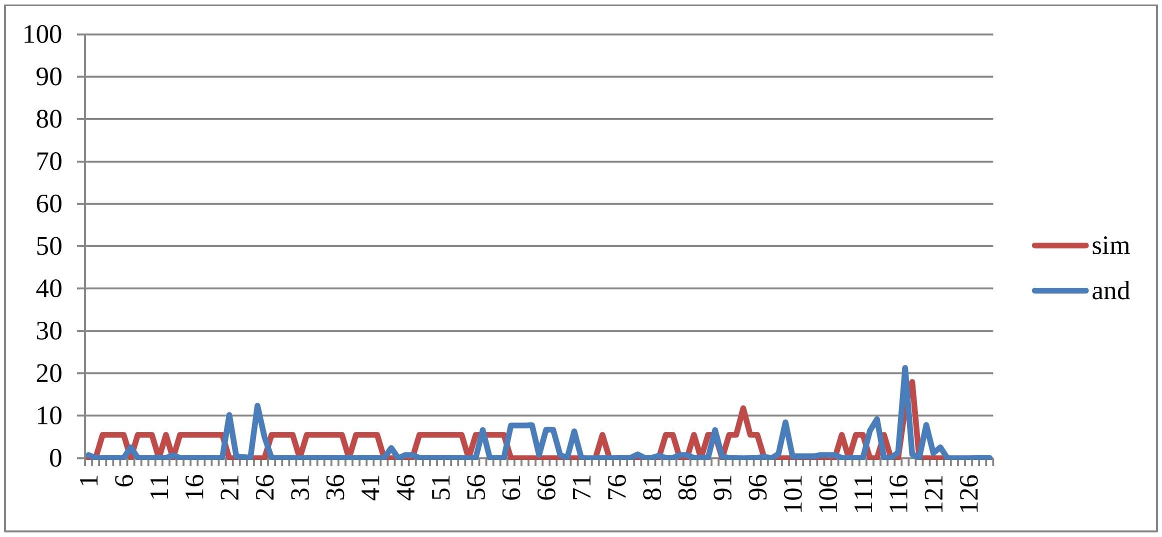


Figure 4-12: Similarity Measurement for Tested Application without Malware

In second graph it is clear that all result are less than 20% which means that this application is free of Malware.

Table 4-4: Experiment Parameters and Output

Method	Is injected (yes/no)	Time (mm:ss)
Androguard	Yes	34:59
Androguard	No	32:34
SimHash	Yes	14:24
SimHash	No	10:18

Table shows that our proposed method takes time less than Androguard to generate database of signatures and detect Malware this also insure the efficiency of our proposed algorithms

4.4 Summary

In this shows the used methodology for thesis, First we describe how we ran Androguard project and how we modify it to use SimHash algorithm as build in tool, then we prove the feasibility of idea by experiments. The first experiment compare between Androguard and thesis model by applying two of them on a Malware data set, the results prove that proposed method is faster than Androguard with similar results and time distribution. Finally the second experiment detected a Malware application and compare detection evidence with Androguard.

5 Conclusion and Future Work

Today life activities for all people depend on latest technologies which provide fast and available communication and production services, smartphones are one of those technologies, it is used everywhere, by everyone, for almost purposes. The wide use of smartphones applications leads for wide growth in Malwares applications which aims to threat users.

Android is the most shared OS for smart phones and it has the biggest number of Malwares , In this thesis an Introduction about Android has been discussed from Android website we talked about Android architecture , components and activity life cycle; Thesis talked about Malwares in general and Malware in Android applications with more details which has been extracted from McAfee 1st Quarter 2013 report , In addition this thesis summarized a group of related work in the topic of Android Malwares detection and SimHash algorithm .

Our contributing was to solve the problem of high cost Android Malware detection by providing a new modified method which saves computation cost using SimHash algorithm. The research main idea is to enhance the similarity measurement to find Malwares in Androguard tool by using SimHash algorithm. This idea is close to one presented in [32] but they use SimHash to enhance detection of clones codes in large system which lead to unresolved bug or maintenance related problems by increasing the risk of update anomalies.

We used SimHash as a part of known tool called Androguard which is mainly a python tool done by VirusTotal project, VirusTotal is a subsidiary of Google, is a free online service that analyzes files and URLs enabling the identification of viruses, worms, trojans and other kinds of malicious content, Androguard uses normalized compression distance algorithm to measure similarity between reversed codes, We replace this algorithm with SimHash algorithm which uses a fingerprint system, instead of comparing the texts directly, it compared their fingerprints, which is really more effective and usable in reality.

To evaluate our idea we passed some experiments. The first experiment compared between Androguard and thesis model by applying two of them on a Malware data set collected from Contagiodump which provides smartphone Malware that infects various smartphone platforms such as Android; A complete dataset of different Malware Android applications (130 different Malware) has been downloaded from Contagiodump as a result research proposed method saved 70% of time comparing with Androguard tool with same time distribution and results similar to it. This refer to using SimHash instead of normal compression to measure distance and saving the time of analyzing reverse codes. Finally the second experiment detected a Malware application called angry bird and compared detection evidences with Androguard.

As a future work enlarge the dataset by immigrate it with other existing Malwares datasets will decrease the false detection. A complete modified Android model with SimHash algorithm can be programmed and released for public users, increasing the dataset and tested applications will be considered to find new model’s bugs. Also users feedback collection will cause more enhancements. In other hand we can utilize the using of SimHash by changing the feature extraction method which can consider the Android code and .xml syntax. Finally changing the tools that used for reverse engineering and combine it with simHash for measure similarity can also generate good results.

6 References

1. Aiman A. Abu Samra, K. Y. (2013). Analysis of Clustering Technique in Android. *2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*.

2. Akamasa Isohara, K. T. (2011). Kernel-based Behavior Analysis for Android Malware Detection. *IEEE Seventh International Conference on Computational Intelligence and Security* .

3. Android. (n.d.). *Activities*. Retrieved 1 23, 2014, from Android Developers: http://developer.android.com/guide/components/activities.html

4. Android. (n.d.). *Application Fundamentals*. Retrieved 1 28, 2014, from Android Developers.

5. Android. (n.d.). *Introduction to Android*. Retrieved 1 23, 2014, from Android Developers: http://developer.android.com/guide/index.html

6. *APKInspector*. (n.d.). Retrieved 6 3, 2014, from APKInspector: https://code.google.com/p/apkinspector/

7. Asaf Shabtai, Y. F. (2010). Automated Static Code Analysis for Classifying Android Applications UsingMachine Learning. *IEEE 2010 International Conference on Computational Intelligence and Security.*

8. B. Dixon, Y. J. (2011). Location based power analysis to detect malicious code in Smartphones. *1st ACM workshop on Security and privacy in Smartphones and mobile device*.

9. B. Lague, D. P. (1997). Assessing the benefits of incorporating function clone detection in a development process. *Proc. ICSM*.

10. Baker, B. S. (1999). A Program for Identifying Duplicated Code. *CSS Interface Proc.*

11. Bryan Dixon, S. M. (2010). On Rootkit and Malware Detection in Smartphones. *IEEE International Conference on Dependable Systems and Networks Workshops (DSN-W)* .

12. Charikar, M. S. (2002). Similarity estimation techniques from rounding algorithms. *Proc. ACM STOC*.

13. D.H. Shih, B. L. (2008). Security aspects of mobile phone virus: a critical survey. *ndustrial Management & Data Systems*.

14. Desnos, A. (2012). Android : Static Analysis Using Similarity Distance. *45th Hawaii International Conference on System Sciences*.

15. *DroidBox*. (n.d.). Retrieved 6 3, 2014, from DroidBox: https://code.google.com/p/droidbox/

16. F. Di Cerbo, A. G. (2011). Detection of malicious applications on android os. *4th international conference on Computational forensics, IWCF'10*.

17. Galopin, T. (n.d.). *A web developer blog*. Retrieved 3 3, 2014, from http://titouangalopin.com/blog/2013/11/simhash-or-the-way-to-compare-quickly-two-datasets

18. Gartner. (2014, 3 14). *Gartner Says Smartphone Sales Grew 46.5 Percent in Second Quarter of 2013 and Exceeded Feature Phone Sales for First Time*. Retrieved 5 31, 2014, from Gartner: http://www.gartner.com/newsroom/id/2573415

19. Google. (n.d.). *Androguard*. Retrieved 3 2014, 4, from Google Code: https://code.google.com/p/androguard/#Papers

20. Heloise Pieterse, M. S. (2012). Android Botnets on the Rise: Trends and Characteristics. *IEEE*.

21. I. Santos, C. L. (2011). Collective classification for unknown malware detection. *in Proceedings of the 6th International Conference on Security and Cryptography (SECRYPT).*

22. I. Santos, F. B.-P. (n.d.). Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences, in press.*

23. *IDA pro*. (n.d.). Retrieved 6 3, 2014, from IDA pro: https://www.hex-rays.com/products/ida/index.shtml

24. InfoSec, S. I. (2011). *Reverse Engineering Of Malware On Android.*

25. Ioannis Charalampopoulos, I. A. (n.d.). A Comparable Study employing WEKA Clustering/Classification Algorithms for Web Page Classification.

26. J. Bergeron, M. D. (2001). Static detection of malicious code in executable programs. *In Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS'01), .*

27. J. Cheng, S. W. (2007). SmartSiren: virus detection and alert for Smartphones. *Proc. International Conference on Mobile Systems, Applications and Services*.

28. Justin Sahs, L. K. (2012). A Machine Learning Approach to Android Malware Detection. *IEE European Intelligence and Security Informatics Conference*.

29. M. Henzinger. (2006). Finding near-duplicate web pages: a large-scale. *Proc. SIGIR*.

30. Mario Frank, B. D. (n.d.). Mining Permission Request Patterns from Android and Facebook Applications.

31. McAfee. (2013). *McAfee Threats Report: First Quarter 2013.* McAfee .

32. Md. Sharif Uddin, C. K. (2011). On the Effectiveness of Simhash for Detecting Near-Miss Clones in Large Scale Software Systems. *IEEE 18thWorking Conference on Reverse Engineering* .

33. Mila. (2014, 3 10). *Contagi Mobile*. Retrieved from (http://contagiominidump.blogspot.com.au/),

34. *Mobile Sandbox*. (n.d.). Retrieved 6 3, 2014, from Mobile Sandbox: http://mobilesandbox.org/

35. MoutazAlazab, V. L. (2012). Analysis of Malicious and Benign Android Applications. *IEEE 32nd International Conference on Distributed Computing Systems Workshops* .

36. NetApplications. (2014). *Annual Survey Report.*

37. P. Berthomé, T. F.-F. (2012). Repackaging Android Applications for Auditing Access to Private Data. *IEEE Seventh International Conference on Availability, Reliability and Security*.

38. Pearson, K. (1895). *Contributions to the Mathematical Theory of Evolution.* Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences.

39. Rafael Fedler, C. B. (2012). Android OS Security:Risks and Limitations. *Fraunh ofer Research Institution for Applied and Integrate d Security*.

40. Sharp, R. (2013). An Introduction to Malware. Technical university of Denmark .

41. Takamasa Isohara, K. T. (2011). Kernel-based Behavior Analysis for Android Malware Detection. *IEEE Seventh International Conference on Computational Intelligence and Security*.

42. Te-En Wei, C.-H. A.-M.-T.-J. (2012). Android Malware Detection via a Latent Network Behavior Analysis. *IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications* .

43. Thomas Dietterich, C. B. (2010). Introduction to Machine Learning Second Edition. The MIT Press.

44. VirusTotal. (n.d.). *VirusTotal*. Retrieved 3 2014, 4, from about-VirusTotal : https://www.virustotal.com/en/about

45. W. Enck, P. G.-G. (2010). Taintdroid: An information-flow tracking system for realtime privacy monitoring on Smartphones. *9th USENIX conference on Operating systems design and implementation*.

46. ZheMin Yang, M. Y. (2012). “LeakMiner: Detect information leakage on Android with static taint analysis. *IEEE Third World Congress on Software Engineering*.

7 Appendices

7.1 Appindex1: Androguard Code.

- Androguard,py

```
import sys, xml.dom.minidom, re, random, string, os

PATH_INSTALL = "./"

sys.path.append(PATH_INSTALL + "/core")
sys.path.append(PATH_INSTALL + "/core/bytecodes")
sys.path.append(PATH_INSTALL + "/core/bytecodes/libdvm")
sys.path.append(PATH_INSTALL + "/core/predicates")
sys.path.append(PATH_INSTALL + "/core/analysis")
sys.path.append(PATH_INSTALL + "/core/analysis/libsign")
sys.path.append(PATH_INSTALL + "/core/vm")
sys.path.append(PATH_INSTALL + "/core/wm")
sys.path.append(PATH_INSTALL + "/core/protection")
sys.path.append(PATH_INSTALL + "/classification")

import bytecode, jvm, dvm, apk, androconf, analysis, opaque
from androconf import error

VM_INT_AUTO = 0
VM_INT_BASIC_MATH_FORMULA = 1
VM_INT_BASIC_PRNG = 2
INVERT_VM_INT_TYPE = { "VM_INT_AUTO" : VM_INT_AUTO,
                       "VM_INT_BASIC_MATH_FORMULA" :
VM_INT_BASIC_MATH_FORMULA,
                       "VM_INT_BASIC_PRNG" : VM_INT_BASIC_PRNG
                     }
class VM_int :
    """VM_int is the main high level Virtual Machine object to
protect a method by remplacing all integer contants

       @param andro : an L{Androguard} / L{AndroguardS} object to
have full access to the desired information
       @param class_name : the class of the method
       @param method_name : the name of the method to protect
       @param descriptor : the descriptor of the method
       @param vm_int_type : the type of the Virtual Machine
    """
    def __init__(self, andro, class_name, method_name, descriptor,
vm_int_type) :
        import vm

        method, _vm = andro.get_method_descriptor(class_name,
method_name, descriptor)
```

```
        code = method.get_code()

        # LOOP until integers constant !
        iip = True
        while iip == True :
            idx = 0
            end_iip = True
            for bc in code.get_bc().get() :
                if bc.get_name() in _vm.get_INTEGER_INSTRUCTIONS() :
                    if vm_int_type == VM_INT_BASIC_MATH_FORMULA :
                        vi = vm.VM_int_basic_math_formula(
class_name, code, idx )
                    elif vm_int_type == VM_INT_BASIC_PRNG :
                        vi = vm.VM_int_basic_prng( class_name, code,
idx )
                    else :
                        raise("oops")

                    for new_method in vi.get_methods() :
                        _vm.insert_direct_method(
new_method.get_name(), new_method )
                    vi.patch_code()

                    end_iip = False

                    break
                idx += 1

            # We have patched zero integers, it's the end my friend
!
            if end_iip == True :
                iip = False

        method.show()

class WM :
    def __init__(self, andro, class_name, wm_type) :
        if wm_type == [] :
            raise("....")

        import wm
        self._w = wm.WM( andro.get_vm(), class_name, wm_type,
andro.get_analysis() )

    def get(self) :
        return self._w

class WMCheck :
    def __init__(self, andro, class_name, input_file) :
        fd = open(input_file, "rb")
        buffxml = fd.read()
        fd.close()
        document = xml.dom.minidom.parseString(buffxml)

        w_orig = wm.WMLoad( document )
        w_cmp = wm.WMCheck( w_orig, andro, andro.get_analysis() )

def OBFU_NAMES_GEN(prefix="") :
    return prefix + random.choice( string.letters ) + ''.join([
random.choice(string.letters + string.digits) for i in range(10 - 1)
] )

OBFU_NAMES_FIELDS = 0
OBFU_NAMES_METHODS = 1
class OBFU_Names :
    """
       OBFU_Names is the object that change the name of a field or a
method by a random string, and resolving
       dependencies into other files

       @param andro : an L{Androguard} object to have full access to
the desired information, and represented a pool of files with the
same format
       @param class_name : the class of the method/field (a python
regexp)
       @param name : the name of the method/field (a python regexp)
       @param descriptor : the descriptor of the method/field (a
python regexp)
       @param obfu_type : the type of the obfuscated (field/method)
(OBFU_NAMES_FIELDS, OBFU_NAMES_METHODS)
       @param gen_method : a method which generate random string
    """
    def __init__(self, andro, class_name, name, descriptor,
obfu_type, gen_method=OBFU_NAMES_GEN) :
        if obfu_type != OBFU_NAMES_FIELDS and obfu_type !=
OBFU_NAMES_METHODS :
            raise("ooops")

        re_class_name = re.compile(class_name)
        re_name = re.compile(name)
        re_descriptor = re.compile(descriptor)

        if obfu_type == OBFU_NAMES_FIELDS :
            search_in = andro.gets("fields")
        elif obfu_type == OBFU_NAMES_METHODS :
            search_in = andro.gets("methods")

        depends = []

        # Change the name of all fields/methods
        for fm in search_in :
            if re_class_name.match( fm.get_class_name() ) :
                if re_name.match( fm.get_name() ):
                    if re_descriptor.match( fm.get_descriptor() ) :
                        _, _vm = andro.get_method_descriptor(
fm.get_class_name(), fm.get_name(), fm.get_descriptor() )
                        old_name = fm.get_name()
                        new_name = gen_method()

                        # don't change the constructor for a .class
file
                        if obfu_type == OBFU_NAMES_METHODS :
                            _, _vm = andro.get_method_descriptor(
fm.get_class_name(), fm.get_name(), fm.get_descriptor() )
                            if _vm.get_type() == "JVM" and old_name
!= "<init>" :
                                fm.set_name( new_name )
                                depends.append( (fm, old_name) )
                        elif obfu_type == OBFU_NAMES_FIELDS :
                            fm.set_name( new_name )
                            depends.append( (fm, old_name) )

        # Change the name in others files
        for i in depends :
            for _vm in andro.get_vms() :
                if obfu_type == OBFU_NAMES_FIELDS :
                    _vm.set_used_field( [ i[0].get_class_name(),
i[1], i[0].get_descriptor() ], [ i[0].get_class_name(),
i[0].get_name(), i[0].get_descriptor() ] )
                elif obfu_type == OBFU_NAMES_METHODS :
                    _vm.set_used_method( [ i[0].get_class_name(),
i[1], i[0].get_descriptor() ], [ i[0].get_class_name(),
i[0].get_name(), i[0].get_descriptor() ] )

class BC :
    def __init__(self, bc) :
        self.__bc = bc

    def get_vm(self) :
        return self.__bc

    def get_analysis(self) :
        return self.__a

    def analyze(self) :
        self.__a = analysis.VMAnalysis( self.__bc,
code_analysis=True )

    def _get(self, val, name) :
        l = []
        r = getattr(self.__bc, val)(name)
        for i in r :
            l.append( i )
        return l
    def _gets(self, val) :
        l = []
        r = getattr(self.__bc, val)()
        for i in r :
            l.append( i )
        return l

    def gets(self, name) :
        return self._gets("get_" + name)

    def get(self, val, name) :
        return self._get("get_" + val, name)

    def insert_direct_method(self, name, method) :
        return self.__bc.insert_direct_method(name, method)

    def insert_craft_method(self, name, proto, codes) :
        return self.__bc.insert_craft_method( name, proto, codes)

    def show(self) :
        self.__bc.show()

    def pretty_show(self) :
        self.__bc.pretty_show( self.__a )

    def save(self) :
        return self.__bc.save()

    def __getattr__(self, value) :
        return getattr(self.__bc, value)

PROTECT_VM_AUTO = "protect_vm_auto"
PROTECT_VM_INTEGER = "protect_vm_integer"
PROTECT_VM_INTEGER_TYPE = "protect_vm_integer_type"

class Androguard :
    """Androguard is the main object to abstract and manage
differents formats

       @param files : a list of filenames (filename must be
terminated by .class or .dex)
       @param raw : specify if the filename is in fact a raw buffer
(default : False) #FIXME
    """
    def __init__(self, files, raw=False) :
        self.__files = files

        self.__orig_raw = {}
        for i in self.__files :
            self.__orig_raw[ i ] = open(i, "rb").read()

        self.__bc = []
        self._analyze()

    def _iterFlatten(self, root):
        if isinstance(root, (list, tuple)):
            for element in root :
                for e in self._iterFlatten(element) :
                    yield e
        else:
            yield root

    def _analyze(self) :
        for i in self.__files :
            #print "processing ", i
            if ".class" in i :
                bc = jvm.JVMFormat( self.__orig_raw[ i ] )
            elif ".jar" in i :
                x = jvm.JAR( i )
                bc = x.get_classes()
            elif ".dex" in i :
                bc = dvm.DalvikVMFormat( self.__orig_raw[ i ] )
            elif ".apk" in i :
                x = apk.APK( i )
                bc = dvm.DalvikVMFormat( x.get_dex() )
            else :
                ret_type = androconf.is_Android( i )
                if ret_type == "APK" :
                    x = apk.APK( i )
                    bc = dvm.DalvikVMFormat( x.get_dex() )
                elif ret_type == "DEX" :
                    bc = dvm.DalvikVMFormat( open(i, "rb").read() )
                else :
                    raise( "Unknown bytecode" )

            if isinstance(bc, list) :
                for j in bc :
                    self.__bc.append( (j[0], BC( jvm.JVMFormat(j[1])
) ) )
            else :
                self.__bc.append( (i, BC( bc )) )

    def ianalyze(self) :
        for i in self.get_bc() :
            i[1].analyze()

    def get_class(self, class_name) :
        for _, bc in self.__bc :
            if bc.get_class(class_name) == True :
                return bc
        return None

    def get_raw(self) :
        """Return raw format of all file"""
        l = []
        for _, bc in self.__bc :
            l.append( bc._get_raw() )
        return l

    def get_orig_raw(self) :
        return self.__orig_raw

    def get_method_descriptor(self, class_name, method_name,
descriptor) :
        """
           Return the specific method

           @param class_name : the class name of the method
           @param method_name : the name of the method
           @param descriptor : the descriptor of the method
        """
        for file_name, bc in self.__bc :
            x = bc.get_method_descriptor( class_name, method_name,
descriptor )
            if x != None :
                return x, bc
        return None, None

    def get_field_descriptor(self, class_name, field_name,
descriptor) :
        """
           Return the specific field

           @param class_name : the class name of the field
           @param field_name : the name of the field
           @param descriptor : the descriptor of the field
        """
        for file_name, bc in self.__bc :
            x = bc.get_field_descriptor( class_name, field_name,
descriptor )
            if x != None :
                return x, bc
        return None, None

    def get(self, name, val) :
        """
           Return the specific value for all files

           @param name :
           @param val :
        """
        if name == "file" :
            for file_name, bc in self.__bc :
                if file_name == val :
                    return bc
            return None
        else :
            l = []
            for file_name, bc in self.__bc :
                l.append( bc.get( name, val ) )

            return list( self._iterFlatten(l) )

    def gets(self, name) :
        """
           Return the specific value for all files

           @param name :
        """
        l = []
        for file_name, bc in self.__bc :
            l.append( bc.gets( name ) )

        return list( self._iterFlatten(l) )

    def get_vms(self) :
        return [ i[1].get_vm() for i in self.__bc ]

    def get_bc(self) :
        return self.__bc

    def show(self) :
        """
           Display all files
        """
        for _, bc in self.__bc :
            bc.show()

    def pretty_show(self) :
        """
           Display all files
        """
        for _, bc in self.__bc :
            bc.pretty_show()

    def do(self, fileconf) :
        self.ianalyze()

        fd = open(fileconf, "rb")
        buffxml = fd.read()
        fd.close()

        document = xml.dom.minidom.parseString(buffxml)

        main_path = document.getElementsByTagName( "main_path"
)[0].firstChild.data
```

```
        libs_path = document.getElementsByTagName( "libs_path"
)[0].firstChild.data

        if document.getElementsByTagName( "watermark" ) != [] :
            watermark_item = document.getElementsByTagName(
"watermark" )[0]
            watermark_types = []
            for item in watermark_item.getElementsByTagName( "type"
) :
                watermark_types.append( str( item.firstChild.data )
)
            watermark_output = watermark_item.getElementsByTagName(
"output" )[0].firstChild.data
            print watermark_types, "--->", watermark_output

            fd = open(watermark_output, "w")

            fd.write("<?xml version=\"1.0\"?>\n")
            fd.write("<andro id=\"androguard wm\">\n")
            wms = []
            for i in self.get_bc() :
                for class_name in i[1].get_classes_names() :
                    wm = WM( i[1], class_name, watermark_types )
                    fd.write( wm.get().save() )
            fd.write("</andro>\n")
            fd.close()

        if document.getElementsByTagName( "protect_code" ) != [] :
            import protection

            protect_code_item = document.getElementsByTagName(
"protect_code" )[0]
            protection.ProtectCode( [ i[1] for i in self.get_bc() ],
main_path + libs_path )

#      for item in document.getElementsByTagName('method') :
#         if item.getElementsByTagName( PROTECT_VM_INTEGER
)[0].firstChild != None :
#            if item.getElementsByTagName( PROTECT_VM_INTEGER
)[0].firstChild.data == "1" :
#               vm_type = INVERT_VM_INT_TYPE[
item.getElementsByTagName( PROTECT_VM_INTEGER_TYPE
)[0].firstChild.data ]
#               VM_int( self, item.getAttribute('class'),
item.getAttribute('name'), item.getAttribute('descriptor'), vm_type
)

        if document.getElementsByTagName( "save_path" ) != [] :
            self.save( main_path + document.getElementsByTagName(
"save_path" )[0].firstChild.data )
        else :
            self.save()
```

```
    def save(self, output_dir=None) :
        for file_name, bc in self.get_bc() :
            if output_dir == None :
                output_file_name = file_name
            else :
                output_file_name = output_dir + os.path.basename(
file_name )

            print "[+] [AG] SAVING ... ", output_file_name
            fd = open(output_file_name, "w")
            fd.write( bc.save() )
            fd.close()

class AndroguardS :
    """AndroguardS is the main object to abstract and manage
differents formats but only per filename. In fact this class is just
a wrapper to the main class Androguard

       @param filename : the filename to use (filename must be
terminated by .class or .dex)
       @param raw : specify if the filename is a raw buffer (default
: False)
    """
    def __init__(self, filename, raw=False) :
        self.__filename = filename
        self.__orig_a = Androguard( [ filename ], raw )
        self.__a = self.__orig_a.get( "file", filename )

    def get_orig_raw(self) :
        return self.__orig_a.get_orig_raw()[ self.__filename ]

    def get_vm(self) :
        """
           This method returns the VMFormat which correspond to the
file

           @rtype: L{jvm.JVMFormat} or L{dvm.DalvikVMFormat}
        """
        return self.__a.get_vm()

    def save(self) :
        """
           Return the original format (with the modifications) into
raw format

           @rtype: string
        """
        return self.__a.save()

    def __getattr__(self, value) :
        try :
            return getattr(self.__orig_a, value)
        except AttributeError :
            return getattr(self.__a, value)
```

- Adrodiff.py

```
import sys, xml.dom.minidom, re, random, string, os

PATH_INSTALL = "./"

sys.path.append(PATH_INSTALL + "/core")
sys.path.append(PATH_INSTALL + "/core/bytecodes")
sys.path.append(PATH_INSTALL + "/core/bytecodes/libdvm")
sys.path.append(PATH_INSTALL + "/core/predicates")
sys.path.append(PATH_INSTALL + "/core/analysis")
sys.path.append(PATH_INSTALL + "/core/analysis/libsign")
sys.path.append(PATH_INSTALL + "/core/vm")
sys.path.append(PATH_INSTALL + "/core/wm")
sys.path.append(PATH_INSTALL + "/core/protection")
sys.path.append(PATH_INSTALL + "/classification")

import bytecode, jvm, dvm, apk, androconf, analysis, opaque
from androconf import error

VM_INT_AUTO = 0
VM_INT_BASIC_MATH_FORMULA = 1
VM_INT_BASIC_PRNG = 2
INVERT_VM_INT_TYPE = { "VM_INT_AUTO" : VM_INT_AUTO,
                       "VM_INT_BASIC_MATH_FORMULA" :
VM_INT_BASIC_MATH_FORMULA,
                       "VM_INT_BASIC_PRNG" : VM_INT_BASIC_PRNG
                     }
class VM_int :
    """VM_int is the main high level Virtual Machine object to
protect a method by remplacing all integer contants

       @param andro : an L{Androguard} / L{AndroguardS} object to
have full access to the desired information
       @param class_name : the class of the method
       @param method_name : the name of the method to protect
       @param descriptor : the descriptor of the method
       @param vm_int_type : the type of the Virtual Machine
    """
    def __init__(self, andro, class_name, method_name, descriptor,
vm_int_type) :
        import vm
        method, _vm = andro.get_method_descriptor(class_name,
method_name, descriptor)
        code = method.get_code()

        # LOOP until integers constant !
        iip = True
        while iip == True :
            idx = 0
            end_iip = True
            for bc in code.get_bc().get() :
                if bc.get_name() in _vm.get_INTEGER_INSTRUCTIONS() :
                    if vm_int_type == VM_INT_BASIC_MATH_FORMULA :
                        vi = vm.VM_int_basic_math_formula(
class_name, code, idx )
                    elif vm_int_type == VM_INT_BASIC_PRNG :
                        vi = vm.VM_int_basic_prng( class_name, code,
idx )
                    else :
                        raise("oops")

                    for new_method in vi.get_methods() :
                        _vm.insert_direct_method(
new_method.get_name(), new_method )
                    vi.patch_code()

                    end_iip = False

                    break
                idx += 1

            # We have patched zero integers, it's the end my friend
!
            if end_iip == True :
                iip = False

        method.show()

class WM :
    def __init__(self, andro, class_name, wm_type) :
        if wm_type == [] :
            raise("....")

        import wm
        self._w = wm.WM( andro.get_vm(), class_name, wm_type,
andro.get_analysis() )

    def get(self) :
        return self._w

class WMCheck :
    def __init__(self, andro, class_name, input_file) :
        fd = open(input_file, "rb")
        buffxml = fd.read()
        fd.close()

        document = xml.dom.minidom.parseString(buffxml)

        w_orig = wm.WMLoad( document )
        w_cmp = wm.WMCheck( w_orig, andro, andro.get_analysis() )

def OBFU_NAMES_GEN(prefix="") :
    return prefix + random.choice( string.letters ) + ''.join([
random.choice(string.letters + string.digits) for i in range(10 - 1)
] )

OBFU_NAMES_FIELDS = 0
OBFU_NAMES_METHODS = 1
class OBFU_Names :
    """
       OBFU_Names is the object that change the name of a field or a
method by a random string, and resolving
       dependencies into other files

       @param andro : an L{Androguard} object to have full access to
the desired information, and represented a pool of files with the
same format
       @param class_name : the class of the method/field (a python
regexp)
       @param name : the name of the method/field (a python regexp)
       @param descriptor : the descriptor of the method/field (a
python regexp)
       @param obfu_type : the type of the obfuscated (field/method)
(OBFU_NAMES_FIELDS, OBFU_NAMES_METHODS)
       @param gen_method : a method which generate random string
    """
    def __init__(self, andro, class_name, name, descriptor,
obfu_type, gen_method=OBFU_NAMES_GEN) :
        if obfu_type != OBFU_NAMES_FIELDS and obfu_type !=
OBFU_NAMES_METHODS :
            raise("ooops")

        re_class_name = re.compile(class_name)
        re_name = re.compile(name)
        re_descriptor = re.compile(descriptor)

        if obfu_type == OBFU_NAMES_FIELDS :
            search_in = andro.gets("fields")
        elif obfu_type == OBFU_NAMES_METHODS :
            search_in = andro.gets("methods")

        depends = []

        # Change the name of all fields/methods
        for fm in search_in :
            if re_class_name.match( fm.get_class_name() ) :
                if re_name.match( fm.get_name() ):
                    if re_descriptor.match( fm.get_descriptor() ) :
                        _, _vm = andro.get_method_descriptor(
fm.get_class_name(), fm.get_name(), fm.get_descriptor() )
                        old_name = fm.get_name()
                        new_name = gen_method()

                        # don't change the constructor for a .class
file
                        if obfu_type == OBFU_NAMES_METHODS :
                            _, _vm = andro.get_method_descriptor(
fm.get_class_name(), fm.get_name(), fm.get_descriptor() )
                            if _vm.get_type() == "JVM" and old_name
!= "<init>" :
                                fm.set_name( new_name )
                                depends.append( (fm, old_name) )
                        elif obfu_type == OBFU_NAMES_FIELDS :
                            fm.set_name( new_name )
                            depends.append( (fm, old_name) )

        # Change the name in others files
        for i in depends :
            for _vm in andro.get_vms() :
                if obfu_type == OBFU_NAMES_FIELDS :
                    _vm.set_used_field( [ i[0].get_class_name(),
i[1], i[0].get_descriptor() ], [ i[0].get_class_name(),
i[0].get_name(), i[0].get_descriptor() ] )
                elif obfu_type == OBFU_NAMES_METHODS :
                    _vm.set_used_method( [ i[0].get_class_name(),
i[1], i[0].get_descriptor() ], [ i[0].get_class_name(),
i[0].get_name(), i[0].get_descriptor() ] )

class BC :
    def __init__(self, bc) :
        self.__bc = bc

    def get_vm(self) :
        return self.__bc

    def get_analysis(self) :
        return self.__a

    def analyze(self) :
        self.__a = analysis.VMAnalysis( self.__bc,
code_analysis=True )

    def _get(self, val, name) :
        l = []
        r = getattr(self.__bc, val)(name)
        for i in r :
            l.append( i )
        return l

    def _gets(self, val) :
        l = []
        r = getattr(self.__bc, val)()
        for i in r :
            l.append( i )
        return l

    def gets(self, name) :
        return self._gets("get_" + name)

    def get(self, val, name) :
        return self._get("get_" + val, name)

    def insert_direct_method(self, name, method) :
        return self.__bc.insert_direct_method(name, method)

    def insert_craft_method(self, name, proto, codes) :
        return self.__bc.insert_craft_method( name, proto, codes)

    def show(self) :
        self.__bc.show()

    def pretty_show(self) :
        self.__bc.pretty_show( self.__a )

    def save(self) :
        return self.__bc.save()

    def __getattr__(self, value) :
        return getattr(self.__bc, value)

PROTECT_VM_AUTO = "protect_vm_auto"
PROTECT_VM_INTEGER = "protect_vm_integer"
PROTECT_VM_INTEGER_TYPE = "protect_vm_integer_type"

class Androguard :
    """Androguard is the main object to abstract and manage
differents formats

       @param files : a list of filenames (filename must be
terminated by .class or .dex)
       @param raw : specify if the filename is in fact a raw buffer
(default : False) #FIXME
    """
    def __init__(self, files, raw=False) :
        self.__files = files

        self.__orig_raw = {}
        for i in self.__files :
            self.__orig_raw[ i ] = open(i, "rb").read()

        self.__bc = []
        self._analyze()

    def _iterFlatten(self, root):
        if isinstance(root, (list, tuple)):
            for element in root :
                for e in self._iterFlatten(element) :
                    yield e
        else:
            yield root

    def _analyze(self) :
        for i in self.__files :
            #print "processing ", i
            if ".class" in i :
                bc = jvm.JVMFormat( self.__orig_raw[ i ] )
            elif ".jar" in i :
                x = jvm.JAR( i )
                bc = x.get_classes()
            elif ".dex" in i :
                bc = dvm.DalvikVMFormat( self.__orig_raw[ i ] )
            elif ".apk" in i :
                x = apk.APK( i )
                bc = dvm.DalvikVMFormat( x.get_dex() )
            else :
                ret_type = androconf.is_Android( i )
                if ret_type == "APK" :
                    x = apk.APK( i )
                    bc = dvm.DalvikVMFormat( x.get_dex() )
                elif ret_type == "DEX" :
                    bc = dvm.DalvikVMFormat( open(i, "rb").read() )
                else :
                    raise( "Unknown bytecode" )

            if isinstance(bc, list) :
                for j in bc :
                    self.__bc.append( (j[0], BC( jvm.JVMFormat(j[1])
) ) )
            else :
                self.__bc.append( (i, BC( bc )) )

    def ianalyze(self) :
        for i in self.get_bc() :
            i[1].analyze()

    def get_class(self, class_name) :
        for _, bc in self.__bc :
            if bc.get_class(class_name) == True :
                return bc
        return None
    def get_raw(self) :
        """Return raw format of all file"""
        l = []
        for _, bc in self.__bc :
            l.append( bc._get_raw() )
        return l

    def get_orig_raw(self) :
        return self.__orig_raw

    def get_method_descriptor(self, class_name, method_name,
descriptor) :
        """
           Return the specific method

           @param class_name : the class name of the method
           @param method_name : the name of the method
           @param descriptor : the descriptor of the method
        """
        for file_name, bc in self.__bc :
            x = bc.get_method_descriptor( class_name, method_name,
descriptor )
            if x != None :
                return x, bc
        return None, None

    def get_field_descriptor(self, class_name, field_name,
descriptor) :
        """
           Return the specific field

           @param class_name : the class name of the field
           @param field_name : the name of the field
           @param descriptor : the descriptor of the field
        """
        for file_name, bc in self.__bc :
            x = bc.get_field_descriptor( class_name, field_name,
descriptor )
            if x != None :
                return x, bc
        return None, None

    def get(self, name, val) :
        """
           Return the specific value for all files

           @param name :
           @param val :
        """
        if name == "file" :
            for file_name, bc in self.__bc :
                if file_name == val :
                    return bc

            return None
        else :
            l = []
            for file_name, bc in self.__bc :
                l.append( bc.get( name, val ) )

            return list( self._iterFlatten(l) )

    def gets(self, name) :
        """
           Return the specific value for all files

           @param name :
        """
        l = []
        for file_name, bc in self.__bc :
            l.append( bc.gets( name ) )

        return list( self._iterFlatten(l) )

    def get_vms(self) :
        return [ i[1].get_vm() for i in self.__bc ]

    def get_bc(self) :
        return self.__bc

    def show(self) :
        """
           Display all files
        """
        for _, bc in self.__bc :
            bc.show()

    def pretty_show(self) :
        """
           Display all files
        """
        for _, bc in self.__bc :
            bc.pretty_show()

    def do(self, fileconf) :
        self.ianalyze()

        fd = open(fileconf, "rb")
        buffxml = fd.read()
        fd.close()

        document = xml.dom.minidom.parseString(buffxml)
        main_path = document.getElementsByTagName( "main_path"
)[0].firstChild.data
        libs_path = document.getElementsByTagName( "libs_path"
)[0].firstChild.data

        if document.getElementsByTagName( "watermark" ) != [] :
            watermark_item = document.getElementsByTagName(
"watermark" )[0]
            watermark_types = []
            for item in watermark_item.getElementsByTagName( "type"
) :
                watermark_types.append( str( item.firstChild.data )
)
            watermark_output = watermark_item.getElementsByTagName(
"output" )[0].firstChild.data
            print watermark_types, "--->", watermark_output

            fd = open(watermark_output, "w")

            fd.write("<?xml version=\"1.0\"?>\n")
            fd.write("<andro id=\"androguard wm\">\n")
            wms = []
            for i in self.get_bc() :
                for class_name in i[1].get_classes_names() :
                    wm = WM( i[1], class_name, watermark_types )
                    fd.write( wm.get().save() )
            fd.write("</andro>\n")
            fd.close()

        if document.getElementsByTagName( "protect_code" ) != [] :
            import protection

            protect_code_item = document.getElementsByTagName(
"protect_code" )[0]
            protection.ProtectCode( [ i[1] for i in self.get_bc() ],
main_path + libs_path )

#       for item in document.getElementsByTagName('method') :
#          if item.getElementsByTagName( PROTECT_VM_INTEGER
)[0].firstChild != None :
#             if item.getElementsByTagName( PROTECT_VM_INTEGER
)[0].firstChild.data == "1" :
#                vm_type = INVERT_VM_INT_TYPE[
item.getElementsByTagName( PROTECT_VM_INTEGER_TYPE
)[0].firstChild.data ]
#                VM_int( self, item.getAttribute('class'),
item.getAttribute('name'), item.getAttribute('descriptor'), vm_type
)

        if document.getElementsByTagName( "save_path" ) != [] :
            self.save( main_path + document.getElementsByTagName(
"save_path" )[0].firstChild.data )
```

```
        else :
            self.save()

    def save(self, output_dir=None) :
        for file_name, bc in self.get_bc() :
            if output_dir == None :
                output_file_name = file_name
            else :
                output_file_name = output_dir + os.path.basename(
file_name )

            print "[+] [AG] SAVING ... ", output_file_name
            fd = open(output_file_name, "w")
            fd.write( bc.save() )
            fd.close()

class AndroguardS :
    """AndroguardS is the main object to abstract and manage
differents formats but only per filename. In fact this class is just
a wrapper to the main class Androguard

       @param filename : the filename to use (filename must be
terminated by .class or .dex)
       @param raw : specify if the filename is a raw buffer (default
: False)
    """
    def __init__(self, filename, raw=False) :
        self.__filename = filename
        self.__orig_a = Androguard( [ filename ], raw )
        self.__a = self.__orig_a.get( "file", filename )

    def get_orig_raw(self) :
        return self.__orig_a.get_orig_raw()[ self.__filename ]

    def get_vm(self) :
        """
           This method returns the VMFormat which correspond to the
file

           @rtype: L{jvm.JVMFormat} or L{dvm.DalvikVMFormat}
        """
        return self.__a.get_vm()

    def save(self) :
        """
           Return the original format (with the modifications) into
raw format

           @rtype: string
        """
        return self.__a.save()
    def __getattr__(self, value) :
        try :
            return getattr(self.__orig_a, value)
        except AttributeError :
            return getattr(self.__a, value)
```

- Androlyze.py

```
import sys, os, cmd, threading, code, re

from optparse import OptionParser

from androguard import *
from bytecode import *
from jvm import *
from dvm import *
from apk import *
from analysis import *
from diff import *
from msign import *

import androconf

import IPython.ipapi
from IPython.Shell import IPShellEmbed

from cPickle import dumps, loads

option_0 = { 'name' : ('-i', '--input'), 'help' : 'file : use this
filename', 'nargs' : 1 }
option_1 = { 'name' : ('-d', '--display'), 'help' : 'display the
file in human readable format', 'action' : 'count' }
option_2 = { 'name' : ('-m', '--method'), 'help' : 'display
method(s) respect with a regexp', 'nargs' : 1 }
option_3 = { 'name' : ('-f', '--field'), 'help' : 'display field(s)
respect with a regexp', 'nargs' : 1 }
option_4 = { 'name' : ('-s', '--shell'), 'help' : 'open an
interactive shell to play more easily with objects', 'action' :
'count' }
option_5 = { 'name' : ('-v', '--version'), 'help' : 'version of the
API', 'action' : 'count' }
option_6 = { 'name' : ('-p', '--pretty'), 'help' : 'pretty print !',
'action' : 'count' }
option_7 = { 'name' : ('-t', '--type_pretty'), 'help' : 'set the
type of pretty print (0, 1) !', 'nargs' : 1 }
option_8 = { 'name' : ('-x', '--xpermissions'), 'help' : 'show paths
of permissions', 'action' : 'count' }
```

```
options = [option_0, option_1, option_2, option_3, option_4,
option_5, option_6, option_7, option_8]

def save_session(l, filename) :
    """
        save your session !

        @param l : a list of objects
        @param filename : output filename to save the session
    """
    fd = open(filename, "w")
    fd.write( dumps(l, -1) )
    fd.close()

def load_session(filename) :
    """
        load your session !

        @param filename : the filename where the sessions has been
saved
        @rtype : the elements of your session
    """
    return loads( open(filename, "r").read() )

def interact() :
    ipshell = IPShellEmbed(banner="Androlyze version %s" %
androconf.ANDROGUARD_VERSION)
    ipshell()

def AnalyzeAPK(filename, raw=False) :
    """
        Analyze an Android application and setup all stuff for a
more quickly analysis !

        @param filename : the filename of the Android application or
a buffer which represents the application
        @param raw : True is you would like to use a buffer

        @rtype : return the APK, DalvikVMFormat, and VMAnalysis
objects
    """
    a = APK(filename, raw)

    d = DalvikVMFormat( a.get_dex() )
    dx = VMAnalysis( d )

    ExportVMToPython( d )

    set_pretty_show( 1 )

    return a, d, dx
```

```
def AnalyzeDex(filename, raw=False) :
    """
        Analyze an Android dex file and setup all stuff for a more
quickly analysis !

        @param filename : the filename of the Android dex file or a
buffer which represents the dex file
        @param raw : True is you would like to use a buffe

        @rtype : return the DalvikVMFormat, and VMAnalysis objects
    """
    d = None
    if raw == False :
        d = DalvikVMFormat( open(filename, "rb").read() )
    else :
        d = DalvikVMFormat( raw )

    dx = VMAnalysis( d )

    ExportVMToPython( d )

    set_pretty_show( 1 )

    return d, dx

def sort_length_method(vm) :
    l = []
    for m in vm.get_methods() :
        code = m.get_code()
        if code != None :
            l.append( (code.get_length(), (m.get_class_name(),
m.get_name(), m.get_descriptor()) ) )
    l.sort(reverse=True)
    return l

def main(options, arguments) :
    if options.shell != None :
        interact()

    elif options.input != None :
        _a = AndroguardS( options.input )

        if options.type_pretty != None :
            bytecode.set_pretty_show( int( options.type_pretty ) )

        if options.display != None :
            if options.pretty != None :
                _a.ianalyze()
                _a.pretty_show()
            else :
                _a.show()
        elif options.method != None :
            for method in _a.get("method", options.method) :
                if options.pretty != None :
                    _a.ianalyze()
                    method.pretty_show( _a.get_analysis() )
                else :
                    method.show()

        elif options.field != None :
            for field in _a.get("field", options.field) :
                field.show()

        elif options.xpermissions != None :
            _a.ianalyze()
            perms_access = _a.get_analysis().get_permissions( [] )
            for perm in perms_access :
                print "PERM : ", perm
                for path in perms_access[ perm ] :
                    print "\t%s %s %s (@%s-0x%x)  ---> %s %s %s" % (
path.get_method().get_class_name(), path.get_method().get_name(),
path.get_method().get_descriptor(), \

path.get_bb().get_name(), path.get_bb().start + path.get_idx(), \

path.get_class_name(), path.get_name(), path.get_descriptor())

    elif options.version != None :
        print "Androlyze version %s" % androconf.ANDROGUARD_VERSION

if __name__ == "__main__" :
    parser = OptionParser()
    for option in options :
        param = option['name']
        del option['name']
        parser.add_option(*param, **option)

    options, arguments = parser.parse_args()
    sys.argv[:] = arguments
    main(options, arguments)
```

7.2 Appendix 2: SimHash Code.

- SimHash

```
import re
import Hashlib

class SimHash(object):
    def __init__(self, value):
        self.f = 64
        self.reg = ur'\w+'
        self.value = None

        if isinstance(value, list):
            self.build_by_features(value)
        elif isinstance(value, long):
            self.value = value
        elif isinstance(value, SimHash):
            self.value = value.Hash
        else:
            self.build_by_text(value)

    def _slide(self, content, width=2):
        return [content[i:i+width] for i in xrange(len(content)-
width+1)]

    def _tokenize(self, content):
        ans = []
        content = ''.join(re.findall(self.reg, content))
        ans = self._slide(content)
        return ans

    def build_by_text(self, content):
        features = self._tokenize(content)
        return self.build_by_features(features)

    def build_by_features(self, features):
        print features
        Hashs = [int(Hashlib.md5(w.encode('utf-8')).hexdigest(),
16) for w in features]
        v = [0]*self.f
        for h in Hashs:
            for i in xrange(self.f):
                mask = 1 << i
                v[i] += 1 if h & mask else -1
        ans = 0
        for i in xrange(self.f):
            if v[i] >= 0:
                ans |= 1 << i
        self.value = ans
    def distance(self, another):
        x = (self.value ^ another.value) & ((1 << self.f) - 1)
        ans = 0
        while x:
            ans += 1
            x &= x-1
        return ans
```

7.3 Appendix 3: Test Code.

- Test1 Androguard:

```
import datetime
import sys

from optparse import OptionParser

import androguard, androconf, diff

myfile=open('testty','w')

for x in range(101, 130):


        c = androguard.Androguard(['./apk/'+str(x)+'.apk'])
        c.ianalyze()
      vm1 = c.get_bc()[0][1].get_vm()
        vmx1 = c.get_bc()[0][1].get_analysis()

      for  y in range(x+1, 130):
           a = datetime.datetime.now()
           d = androguard.Androguard(['./apk/'+str(y)+'.apk'])
           d.ianalyze()
           vm2 = d.get_bc()[0][1].get_vm()
           vmx2 = d.get_bc()[0][1].get_analysis()
           dsim = diff.Sim( [ vm1, vmx1 ], [ vm2, vmx2 ] )
           print "compares "+str(x)+" with "+str(y) +"RESULT=
"+str(dsim.get_final_score())
           b = datetime.datetime.now()
           myfile.write("distance between "+str(x)+" and "+str(y)+ "
is "+str(dsim.get_final_score())+" time= "+str(b-a)+"\n")
           if not(d  is None):del d
           if  not (vm2  is None): del vm2
           if  not (vmx2  is None): del vmx2
           if not ( dsim is None ):del dsim


      if not (c  is None ):del c
        if not (vm1  is None): del vm1
```

```
    if not (vmx1 is None): del vmx1
```

- Test2 SimHash:

```
from simHash import SimHash
import datetime
myfile=open('testt','w')
for x in range(22, 130):
    for  y in range(x+1, 130):
        a = datetime.datetime.now()

        f=open("../../apk/"+str(x)+".apk_FILES (2)/classes.dex",
mode='rb')
        test = f.read()
        #test="hddhhfffh";
        f=open("../../apk/"+str(y)+".apk_FILES (2)/classes.dex",
mode='rb')
        test1 = f.read();
        #test1="hddhhfffhf";
        #print lines;
        #a= SimHash(lines).value;
        #b= SimHash('How are you? I am  vey fine. Thanks alot but
what about ypou and father .').value;
        #print '%x' %a
        #print '%x' %b
        xx=SimHash(test)
        yy=SimHash(test1)

        #print xx.value
        #print yy.value

        aa=bin(xx.value)
        bb=bin(yy.value)
        #print aa
        #print bb

        z= xx.value ^ yy.value
        #print bin(z)
        #print x
        print "compares "+str(x)+" with "+str(y)
        b = datetime.datetime.now()
        dd=xx.distance(yy)
        xxx=(1.00000-(dd/64.00000))*100
        myfile.write("similarity between "+str(x)+" and "+str(y)+
" is "+str(xxx)+" time= "+str(b-a)+"\n")
```