Identifying Android Malware Using Machine Learning Based Upon Both Static and Dynamic Features

by

Pacharawit Topark-ngarm

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington
2014

Abstract

A recent report showed that more than half (51.6%) of total phone shipments were smartphones. These devices are as powerful as laptop computers from only a few years ago and are used to browse the Internet, send/receive emails, transfer files, watch, create and transmit multimedia and install applications that add new functionality. As of Q1 2011, the Android smartphone operating system (OS) is the most widely sold operating system worldwide. Unfortunately, the Android malware threat has continuously increased since the first Android malware was reported in 2010. This thesis describes an approach to identify Android malware using a mix of static and dynamic features. The static features are the permissions requested by the application and are obtained from the application itself. Whereas, the dynamic features are extracted from the application at runtime by instrumenting the binary code and executing it in a emulator. This instrumentation approach was developed as part of the work for this thesis. We evaluate the use of the features with a range of machine learning binary classifiers in order to classify an unknown application as either benign or malware.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis Goals 2

1.3 Thesis organisation 3

2 Background and Related Work **5**

2.1 Android 5

2.1.1 Architecture 5

2.1.2 Java Virtual Machine 7

2.1.3 Java Class File Structure 10

2.1.4 The Dalvik Virtual Machine 13

2.1.5 Java vs. Dalvik 14

2.1.6 Android Application Package (APK) 15

2.1.7 The Applications 15

2.2 Malware 16

2.2.1 Virus 16

2.2.2 Worms 17

2.2.3 Trojan Horses 17

2.2.4 Android Malware 17

2.3 Defense 19

2.3.1 Firewalls 19

2.3.2 Anti-Virus 19

2.4 Intrusion Detection System 20

2.4.1 Network-Based Intrusion Detection 20
2.4.2 Host-Based Intrusion Detection 20
2.4.3 Anomaly Detection 21
2.4.4 Honeypots 22
2.5 Android Security 22
2.5.1 Android Security Architecture 22
2.5.2 Android's Bouncer Service 23
2.5.3 Related Work 23
2.6 Summary 26

3 Design **27**
3.1 Motivation 27
3.2 Requirements 28
3.3 Design Decisions 29
3.3.1 Extracting Runtime Features 29
3.3.2 Modularising the Instrumentation Code 30
3.3.3 Hooking to Intercept Intents 31
3.3.4 Static Analysis 31
3.3.5 Format of Output 31
3.4 System Architecture 31
3.5 Hooking at Byte Code Level 33
3.5.1 Parsing 34
3.5.2 Parsing the Instrumentation Class 34
3.5.3 Instrumenting Algorithms 35
3.6 Extract Permission 36
3.7 Formatting for Machine Learning 37
3.8 Machine Learning 38
3.8.1 Training Phase 38
3.8.2 Runtime Phase 40

4 Implementation **41**
4.1 Tools 41

4.1.1 APKtool 42
4.1.2 Smali/Baksmali 42
4.1.3 WEKA 43
4.1.4 Python 45
4.2 Implementation 45
4.2.1 Adding Hooks 45
4.3 Scripts Used to Implement System 48
4.3.1 Disassemble 48
4.3.2 Extract Permissions 48
4.3.3 ARFF File 49
4.3.4 Reassemble 50
4.3.5 Sign APK 51
4.3.6 Install APK 51

5 Experimental Evaluation **53**
5.1 Purpose of Experiments 53
5.2 Data Collection 53
5.2.1 Source of Android Applications 54
5.2.2 Training Set 55
5.2.3 Test Set 55
5.3 Classifiers 55
5.4 Validation 56
5.5 System Environment 57
5.6 Result 58
5.6.1 Accuracy 58
5.6.2 Receiver Operating Characteristic (ROC) 58
5.6.3 AUC 61
5.7 Discussion 61

6 Conclusion **63**
6.1 Contribution 64
6.2 Future Work 65

6.2.1 Greater Feature Detection 65
6.2.2 Monitoring Physical Phones 65

List of Figures

2.1 Android Architecture 6
2.2 Converting Java code into byte code to be run on JVM 8
2.3 Diagrammatic representations of a Java class file[1] 11
2.4 Conversion of .jar to .dex 13
2.5 Permission request screen 24

3.1 System Architecture 32
3.2 Flow chart of training phase 39
3.3 Flow chart of runtime phase 40

4.1 Weka Explorer 44
4.2 Example of Weka result page 44

5.1 Naive Bayes ROC Curve 59
5.2 K-Nearest Neighbours ROC Curve 60
5.3 Decision Tree (J48) ROC Curve 60
5.4 Multi-Layer Perceptron ROC Curve 60
5.5 Support Vector Machine ROC Curve 61

List of Tables

2.1 Common Virus, Worm, and Trojan Horse characteristics . . . 18

5.1 The Experimental Results 59
5.2 Area Under the Curve 61
5.3 Result Comparison with Similar Study 62

Listings

2.1 A Java method that returns an integer value. 14
2.2 A Java byte code for the method in listing 2.1. 14
2.3 A Dalvik byte code for the method in listing 2.1. 14
2.4 Part of permission request declared in AndroidManifest.xml 23
3.1 Example of a .smali file 30
3.2 Snippet of a .smali code 34
3.3 Parsing Algorithm 34
3.4 Parsing Algorithm 35
3.5 Modifying the instrumentation class path. 35
3.6 Algorithm to insert hooks into a specified smali file. 36
3.7 A permission requested in AndroidManifest.xml 37
3.8 Example of *.arff* file. 37
4.1 A snippet of the beginning of a smali code. 42
4.2 A snippet of methods in smali syntax. 42
4.3 A snippet of Python script 45
4.4 A method that has four parameters. 46
4.5 Initialization of an array of object in Smali language. 46
4.6 Inserting the parameters into an array and passing the array to the before method. 46
4.7 APKtool command used to dissembled APK 48
4.8 Part of Python script to search and store the requested permissions 48
4.9 Python script to generate ARFF header 49

4.10 Python script to generate ARFF header 50
4.11 APKtool command used to reassemble APK 50
4.12 Command used to sign APK 51
4.13 Command used to install APK 51

Chapter 1

Introduction

Mobile devices have become an inseparable component of most peoples lives[2], replacing personal computers in terms of the Internet usage by allowing users to check emails, access online banking services, tweet, or use Facebook on such devices. Furthermore, the rapidly growing rich mobile applications with overwhelming user experience, such as maps and GPS functions, make mobile devices more appealing to users. As part of utilizing mobile devices, certain sensitive data such as contact lists, passwords and credit card numbers are stored on these mobile devices. Based upon this scenario, hackers have turned their attention to mobile devices[2] where it is possible to obtain an abundance of their preferred data, whereby security issues are taken less seriously on such devices.

1.1 Motivation

Research and development in this area is important because smartphone has became ubiquitous and powerful. For many people, smartphone is their main or their only device to store sensitive information about themselves. According to a website, smartphone have been shipped one billion units in a single year for the first time and accounted for 55.1% of all mobile phone shipments in 2013[3]. Hackers will target smartphone as much,

if not more, as they target the PC. This study will create another option that can help people to be aware of potential Android malware before installing the application.

1.2 Thesis Goals

The primary goal of the study is be able to identify Android malware with the help of a system that combine machine learning classification method and the number of extracted features from Android APK file. This presents a malware detection method for any unknown Android applications. While most of previous studies extracted features that are based on byte sequence n-grams[4] in this study we evaluate the use of meaningful features from the Android application files such as the requested permissions, framework methods, classes used by the application and dynamic features such as invocation of Android API. In this research, I introduced a system that combines features extracted from Android APK and machine learning classification that can be used to detect potential Android malware without the need of the application source code. A set of goal for this research had been set as followed:

1. Design and prototype a framework for extracting static and dynamic features from a given Android application. The key elements of this system are:

    - Development of a binary rewriting mechanism that allows binary application code to be instrumented to allow monitoring of calls to the Android application programming Interface.
    - Using this mechanism to also monitor the invocation of Android runtime services via Events.
    - Identifying permission requests through static analysis of the code.

2. Evaluating the use of machine learning binary classifiers to identify potential Android malware based upon the features extracted in goal number 1.

1.3 Thesis organisation

The remainder of this thesis is organised as follows:

Chapter 2 presents overview structure of Android operating system and Android application architecture. The detail of Java byte code and Dalvik byte code is described. The chapter also presents how Android application is constructed and transform into APK package file. A brief detail of smartphone malware can also be found in this chapter. Chapter 3 presents the design of Mobile Honeypot system. This chapter discusses the process of instrumenting Android application in order to extract features to create a vector for machine learning classification. The basic machine learning techniques also described in this chapter. Chapter 4 presents the implementation of Mobile Honeypot in detail. This chapter discusses the tools needed to run the experiment and how everything is putting together to create Mobile Honeypot. Chapter 5 presents the result of the experiment. This chapter discusses the source of Android application and Android malware. This chapter so the accurate of the system and discusses about each classifier. Chapter 6 summarises the thesis and discuss future work.

Chapter 2

Background and Related Work

This chapter presents the background and provides a review of related work; in particular the existing solutions proposed to instrument a method.

2.1 Android

Android is a smartphone operating system created by Google Inc.. Since 2011, the Android has become the most widely sold smartphone operating system worldwide[5].

2.1.1 Architecture

The Android architecture is divided into the following four main components. Figure 2.1 shows Android architecture.

- The kernel
- The libraries and Dalvik virtual machine
- The application framework
- The applications

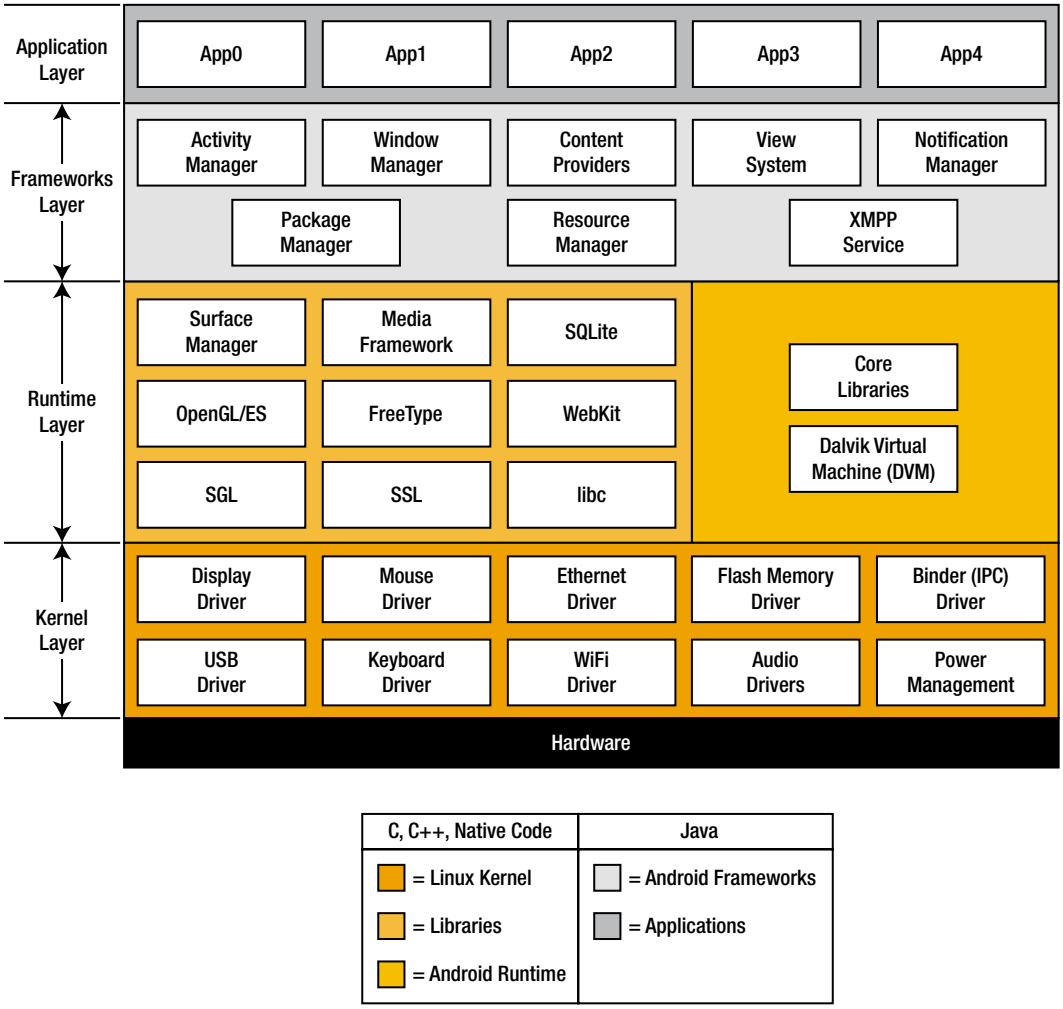


Figure 2.1: Android Architecture

The Kernel

Android runs on top of a Linux 2.6 kernel. The kernel is the first layer of software that interacts with the device hardware. Android kernel will take care of power and memory management, device drivers, process management, networking, and security. In general, end user should not consider modifying or building a new kernel. Although, hardware or device manufacturers will want to modify the kernel to ensure that the operating system works with their specific type of hardware.

The Libraries

The libraries component acts as a translation layer between the kernel and the application framework. Android libraries are written in C/C++ but are shown through a Java API, that means, we can access Android libraries with Java framework.

The runtime component consists of the Dalvik virtual machine that will interact with and run applications. The virtual machine is an important part of the Android operating system and executes system and third-party applications.

2.1.2 Java Virtual Machine

The Java Virtual Machine (JVM)[6] is stack-based. The JVM was developed by Sun Microsystems, Inc., which is now owned by Oracle. The JVM is the basis of the Java platform. It is the component of the technology responsible for its hardware and operating system-independence. It is well known for the small size of its compiled code and its ability to protect users from malicious programs.

The JVM is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulate various memory areas at run-time. The JVM runs inside a Virtual Machine (VM) allowing the Java code to be executed on variety of platforms.

The Java code is stored in .java file. This code contains one or more Java language attributes like classes, methods, variable, and objects. Java (Figure 2.2) is used to compile this code and to generate .class file. Class file is also known as byte code. The Java byte code is an input to the JVM. The JVM reads this code, interprets it and executes the program.

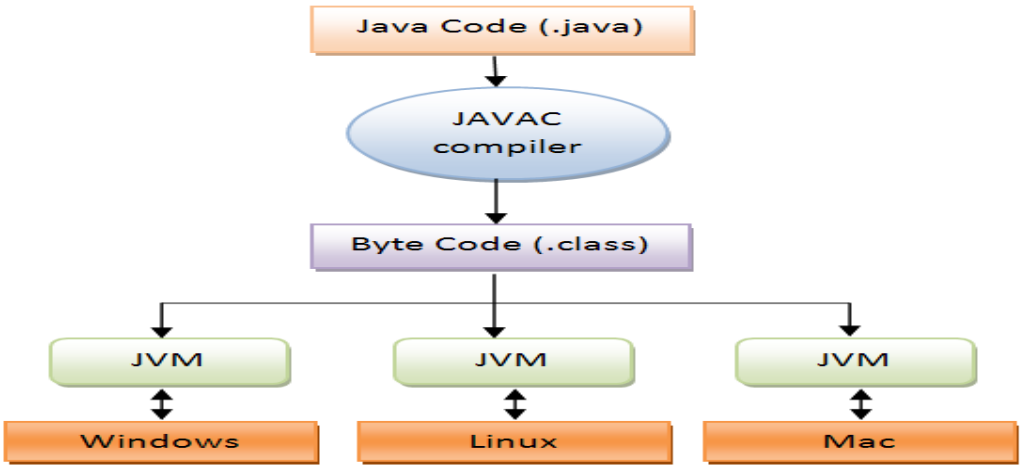


Figure 2.2: Converting Java code into byte code to be run on JVM

The JVM has four registers that are responsible for managing the stack[7]. Since the **registers** of the JVM are similar to the registers in our computer, the VM is stack-based and its registers are not used for passing or receiving arguments. In Java, registers hold the machine's state, and are updated after each line of byte code is executed to maintain that state. The following four registers hold the state of the VM:

1. **Frame**, the reference frame that contains a pointer to the execution environment of the current method.

2. **Optop**, the operand top that contains a pointer to the top of the operand stack, and is used to evaluate arithmetic expressions.

3. **PC**, the program counter that contains the address of the next byte code to be executed.

4. **Vars**, the variable register that contains a pointer to local variables.

The JVM uses an **operand stack** to supply parameters to methods and operations, and to receive results back from them. All byte code instructions take operands from the stack, operate on them, and return results to the stack. Like registers in the VM, the operand stack is 32 bits wide.

Each method in our Java program has a stack frame associated with it. The **stack frame** holds the state of the method with three sets of data: the local variables, the execution environment, and the operand stack. Although the sizes of the local variable and the execution environment data sets are always fixed at the start of the method call, the size of the operand stack changes as the method's byte code instructions are executed. The 64-bit numbers are not guaranteed to be 64-bit aligned as the Java stack is 32 bits wide.

The **execution environment** is maintained within the stack as a data set, and is used to handle dynamic linking, normal method returns, and exception generation. In order to handle dynamic linking, the execution environment contains symbolic references to methods and variables for the current method and current class. These symbolic calls are translated into actual method calls through the dynamic link to a symbol table.

Whenever a method completes normally, a value is returned to the calling method. The execution environment handles normal method returns by restoring the registers of the caller and incrementing the program counter of the caller to skip the method call instruction. Execution of the program then continues in the calling method's execution environment.

If an execution of the current method completes normally, a value is returned to the calling method. This occurs when the calling method executes a return instruction appropriate to the return type. If the calling method executes a return instruction that is not appropriate to the return type, the method throws an exception or an error. Errors that can occur include dynamic linkage failure, such as a failure to find a class file, or run-time errors, such as a reference outside the bounds of an array. When

errors occur, the execution environment generates an exception.

Java's **method area** is similar to the compiled code areas of the run-time environments used by other programming languages. It stores byte code instructions that are associated with methods in the compiled code, and the symbol table the execution environment needs for dynamic linking. Any debugging or additional information that might need to be associated with a method is stored in this area.

Each program running in the Java run-time environment has a **garbage-collected heap** assigned to it as instances of class objects are allocated from this heap, another word for the heap is **memory allocation pool**. By default, the heap size is set to 1MB on most systems. Although the heap is set to a specific size at the start of a program, it can grow, when new objects are allocated. To ensure that the heap does not get too large, the unused objects are automatically reallocated or garbage-collected by the JVM.

2.1.3 Java Class File Structure

A Java class file is consist of 10 basic sections[8] as shown in figure 2.3. The length of the Java class is not known before it gets loaded. There are variable length sections such as constant pool, methods, and attributes. These sections are organized in such a way that they are prefaced by their size or length. This way JVM knows the size of variable length sections before actually loading them.

The above diagram depicts that a Java class file is divided into different components such as magic, version, constant pool, access flags, this class, super class, interfaces, fields, methods, and attributes. The data written in a class file is kept at one byte aligned and is tightly packed. This helps in making class file compact. The order of different sections in a Java class file is strictly defined so that the JVM knows what to expect in a class file and the order of loading different components. The following provides a detailed information about the class files component.

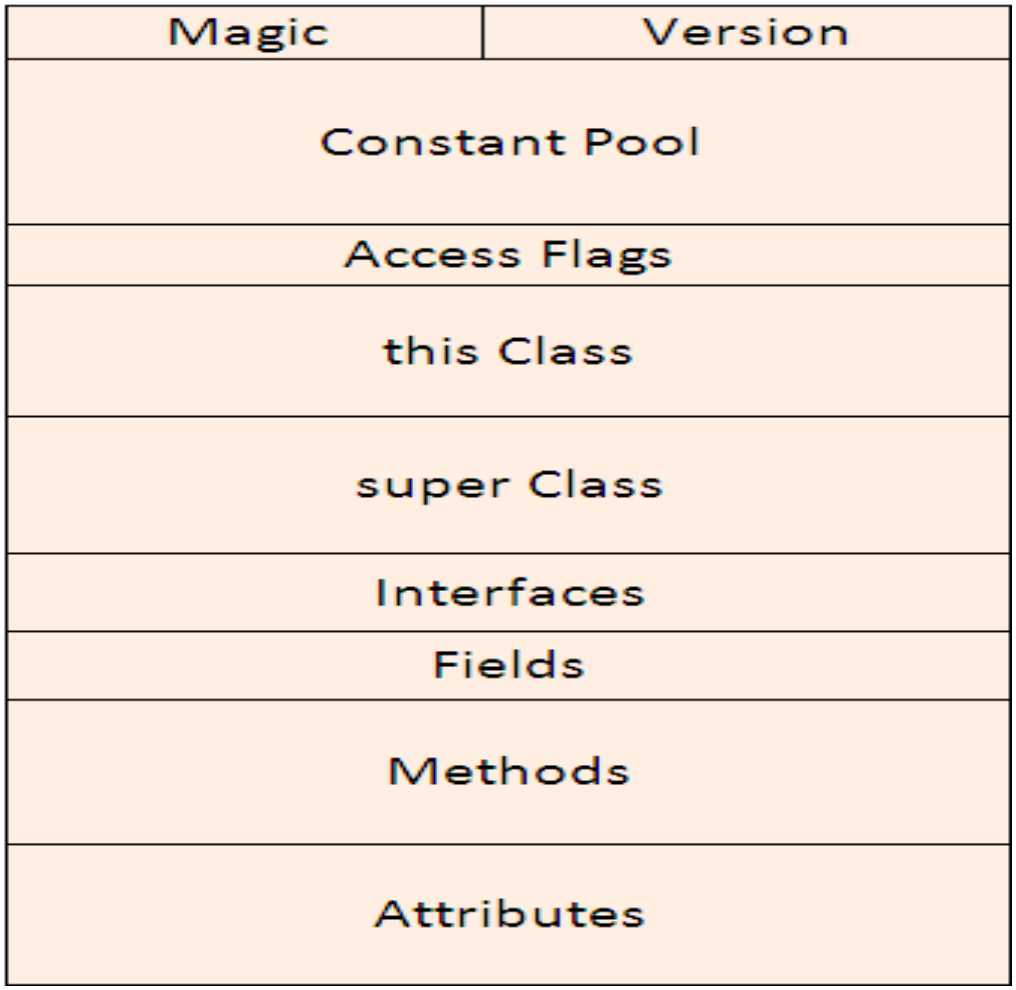


Figure 2.3: Diagrammatic representations of a Java class file[1]

Magic number: is used to uniquely identify the format and to distinguish it from other formats. The first four bytes of the class file are **0xCAFEBABE**. The next four bytes of the class file contain major and minor **version numbers**. This number allows the JVM to verify and identify the class file. If the number is greater than what JVM can load, the class file will be rejected.

Constant pool: all the constants related to the class or an interface will get stored in the constant pool. The constant includes class names, variable names, interface names, method names and signature, final variable values, string literals etc.

Access flags: follows the constant pool. It is a two byte entry that indicates whether the file defines a class or an interface, whether it is public or abstract or final in case it is a class.

This class: is a two byte entry that points to an index in the constant pool. In the above diagram, this class has a value 0x0007 which is an index in constant pool.

Super Class: is the next two bytes after this class. Similar to this class, the value of two bytes is a pointer that points to the constant pool which has entry for super class of the class.

Interfaces: all the interfaces that are implemented by the class (or interface) are defined in the file goes in the interface section of a class file. Starting at two bytes of the interface section is the count that provides information about the total number of interfaces being implemented.

Fields: a field is an instance or a class level variable (property) of the class or interface. The fields section contains only those fields that are defined by the class or an interface of the file and not those which are inherited from the super class or super interface.

Methods: the methods component hosts, that is, the methods that are explicitly defined by this class, not any other methods that may be inherited from the super class.

Attribute section: contains several attribute about the class file, such

as one of the attributes is the source code attribute which reveals the name of the source file from which this class file was compiled.

2.1.4 The Dalvik Virtual Machine

Dan Bornstein named Dalvik after a small fishing village in Iceland. The Dalvik VM[9] was create in order to allow Android application executed on devices with very limited resources. Smartphone is such device because because they are limited by processing power, the amount of memory available, and a short battery life. The Dalvik VM executes .dex files. A .dex file is made by taking the compiled Java .class or .jar files and consolidating all the constants and data within each .class file into a shared constant pool. The dx tool, which comes with the Android SDK, performs this conversion. After conversion, .dex files will have a significantly smaller file size. Figure 2.4 shows how dx tool convert .jar file to .dex file.

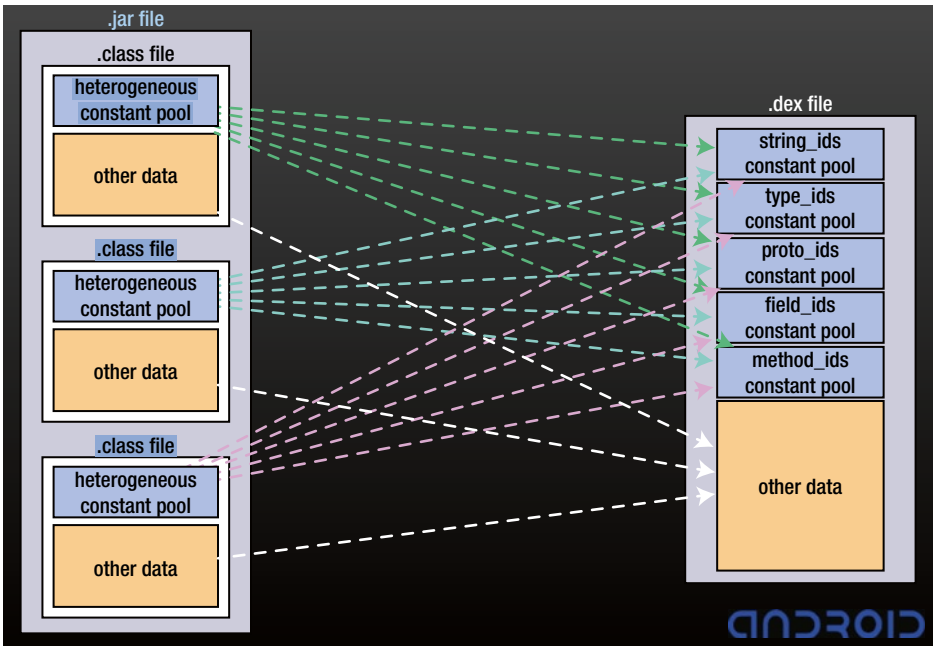


Figure 2.4: Conversion of .jar to .dex

2.1.5 Java vs. Dalvik

The Dalvik virtual machine has a register-based architecture; whereas, the Java virtual machine has a stack-based architecture. Therefore, JVM keeps track of all its variables by using the stack. Operations are then called to perform tasks on the stack. The Java static method in this example takes an integer parameter, and then returns an integer which is the given number plus a fixed number.

```
public static int addConst(int val) {
        return val + 123456;
}
```

Listing 2.1: A Java method that returns an integer value.

```
public static int addConst(int);
        [max_stack=2, max_locals=1, args_size=1]
        0: iload_0
        1: ldc #int 123456
        3: iadd
        4: ireturn
```

Listing 2.2: A Java byte code for the method in listing 2.1.

At the beginning of the Java byte code, it defines the stack size in listing 2.1; the first instruction of the Java byte code (line 0) loads the integer variable (var) onto the stack. The second line (line 1) pushes the constant integer (123456) onto the stack. The following line (line 3) pops the two integers, adds them, and pushes the result back onto the stack. The final line (line 4) returns the final result of the method.

```
public static int addConst(int);
        [regs=2, ins=1, outs=0]
        0: const v0, #0x1E240
        1: add-int/2addr v0, v1
        2: return v0
```

Listing 2.3: A Dalvik byte code for the method in listing 2.1.

The Dalvik byte code is a register-based as the Dalvik byte code example above shows, as it defines the size of the register in the second line. In listing 2.3 the first instruction of the Dalvik byte code (line 0) moves the given constant integer into the specified register (v0). The second instruction (line 1) performs the addition operation on the two source registers,

storing the result in the first source register. The final line (line 2) returns the result that was stored in the first source register (v0). The DVM uses different operation codes (opcodes) structure than the JVM[10].

2.1.6 Android Application Package (APK)

The Android Application Package (APK) file is the file format used to distribute and install application software and middleware onto Google's Android operating system. To make an APK file, a program for Android is first compiled, and then all of its parts are packaged into one file. This holds all of that program's codes (such as classes.dex files), resources, assets, certificates, and manifest file.

DEX file

Android programs are compiled into .dex (Dalvik Executable) files, which are in turn zipped into a single .apk file on the device. .dex files can be created by automatically translating compiled applications written in the Java programming language[11].

Manifest file

Android Manifest file (*AndroidManifest.xml*) is an XML file required by every Android application[12]. The meaning of the manifest file is to describe the application's package name, version, permission required, components (activities, intent filters, and services), imported libraries, and the various activities, and etc.

2.1.7 The Applications

The application component of the Android operating system is the closest to the end user. This section is where the Contacts, Phone, Messaging, and third party apps in. A complete app will execute in this space by using

the API libraries and the Dalvik VM. Even though every component of the Android operating system can be modified, we can only have direct control over our own applications security.

2.2 Malware

Malware is a short word for malicious and software[13], so its software written with malicious intention. There are many techniques that the attackers used to spread the malware. Some of the well known techniques are code injection, file transport, exploit, or boot sector corruption. File injection is the exploitation of a computer bug that is caused by processing invalid data. Code injection can be used by attacker to inject code into a computer program to change the course of execution. For example, code injection is used by some Computer worms to propagate. An exploit is a piece of software, a chunk of data, or sequence of commands that take advantage of a bug, glitch or vulnerability in order to cause unintended or unanticipated behavior to occur on computer software or hardware. This frequently includes such things as violently gaining control of a computer system or allowing privilege escalation or a denial of service attack. Malware can harm the compromised device in many ways. There are three main categories of malicious software: virus, worm, and Trojan horses as summarised in table 2.1.

2.2.1 Virus

A virus is a computer program that can copy itself and infect a computer without the permission or knowledge of the owner. A virus mostly comes in executable file. If the user executes this file the virus processes its malicious commands, which can be almost everything the OS allows with the same privileges as the user.

2.2.2 Worms

A worm can often spread without user interaction. Depending on the OS, this can operate with the same permission as the user. Once started, it searches for infectable victims in range. If a victim is found, it uses an exploit to attach itself to the victim and then repeats this behavior. Sometimes worms drop other malware that can be backdoors that allow remote access. Malicious programs installed that way can make the victim vulnerable to a remote triggered Denial of Service (DoS) attacker.

2.2.3 Trojan Horses

A trojan horse is a program that is disguised as a popular application in order to persuade a user to execute or install it. A trojan often acts as a backdoor, contacting a controller which can then have unauthorized access to the affected computer. A trojan is usually disguised itself by choosing a well-known name like from a popular game and placing the malware for download on a web page or file sharing tool.

2.2.4 Android Malware

The Lookout Mobile Threat Report[14] gives a good summary on how Android malware emerged. When looking at the evolution of malware for mobile phones, the first feature of malware was to send short messages to premium rate numbers or call such numbers[15]. The main incentive here is that the attacker can easily gain money by deploying such methods. Smartphone typically has a connection to the Internet all the time, the next logical step for mobile malware was to develop botnet capabilities. A big step in the Android malware evolution was the utilization of privilege escalation exploits. If the application has root level access to the system, it can use all resources of the system. This allows the application to install other applications, which use arbitrary number of permissions, without

Table 2.1: Common Virus, Worm, and Trojan Horse characteristics

	Malicious Property	Vector	Payload
Virus	need host / require user interaction	file transport, file injection, exploit	replication, variant
Worm	independent program / no user interaction required	exploit	replication, remote access
Trojan Horse	program with hidden agenda/ require user interaction	file transport, exploit	remote access, destructive functionality

the knowledge of the user.

2.3 Defense

In this section, we review the existing defenses against malware.

2.3.1 Firewalls

Firewalls primarily consist of packet filters and/or proxy servers. A packet filter is a component that can restrict network flow based on the information found in the TCP/IP header. Once network flow is permitted, a packet filter does not provide any protection against the data contained in this network flow. However, a packet filter can block access on a particular server port, which would effectively block a particular type of client from connecting to particular types of servers, at the expense of availability of the service. Alternatively, a packet filter can allow for more fine grained control of access. It can prevent access to malicious servers by blocking network flow to these servers.

2.3.2 Anti-Virus

Antivirus software is another defense mechanism. The major of such mechanism relies on up-to-date malware signature database to detect malware[16].The early versions were highly focused to detect just particular types of viruses. Shortly after, first-generation scanners appeared that were able to identify viruses based on simple string matching. Antivirus software initially was tasked with identifying viruses and disinfecting the infected files. The scanning techniques, as a result, were highly specialized to concentrate on binary data within executable files. In addition, antivirus software uses emulators to identify stealth viruses and heuristics to identify unknown viruses. In recent years, antivirus software has started to focus on identification of exploits found on web pages as well. Nevertheless, a recent

evaluation of available antivirus software has revealed that they are quite ineffective in detecting malware[17] probably due to the historical focus on identifying malicious binary data.

2.4 Intrusion Detection System

An Intrusion detection system (IDS) is a network security device that monitors network and/or system activities for malicious or unwanted behavior. There several types of Intrusion Detection Systems.

2.4.1 Network-Based Intrusion Detection

A network-based IDS (NIDS) looks for attack signatures in network traffic[18]. Typically, a network adaptor running in promiscuous mode monitors and analyzes all traffic in real-time as it travels across the network. The attack recognition module uses network packets as the data source. There are three common techniques for recognizing attack signatures: pattern, expression or bytecode matching, frequency or threshold crossing, and correlation of lesser events. Snort is a popular NIDS developed in the open-source community.

2.4.2 Host-Based Intrusion Detection

A host-based IDS (HIDS) looks for attack signatures in log files of hosts[19]. It can also verify the checksums of key system files and executables at regular intervals. Some products can use regular-expressions to refine attact signatures (e.g. passwd program executed AND .rhosts file changed). Some product listen to port activity and generate alerts when specific ports are accessed, providing limited NIDS capability. There is a trend towards host-based intrusion detection. The most effective IDSs combine NIDS and HIDS.

Due to the near real-time nature of IDS alerts, and IDS can be used as a response tool, but automated responses are not without dangers. An attacker might trick the IDS to respond, with the response aimed at an innocent target (e.g. by spoofing the source IP address). Users can be locked out their accounts because of false positives. Repeated email notifications become a denial-of service attack on the administrators email account.

2.4.3 Anomaly Detection

Statistical anomaly detection (or behavior-based detection) uses statistical techniques to detect potential intrusions. First, the 'normal' behavior is defined as a baseline. During operation, a statistical analysis of the data monitored is performed and the deviation from the baseline is measured. If a threshold is exceeded, an alarm is issued. This type of IDS does not need to know about security vulnerabilities in a particular system. The baseline defines normality. So, there is a chance to detect new attacks without having to update a knowledge base.

On the other hand, anomaly detection detects just anomalies. Suspicious behavior does not always define as an intrusion. For example, a number of failed login attempts could be due to an attack or to the administrator forgot the password. There are some problems that we need to point out. Attacks are not always anomalies especially when the baseline is adjusted dynamically and automatically. A careful attacker might just gradually shift 'normality' over time until his planned attack no longer generates an alarm. We have to be concerned about *false positives* (false alarms) when an attack is identified but none is taking place, and *false negatives* when an attack is missed because it acts within the range of normal behavior.

2.4.4 Honeypots

Honeypot is a trap set to detect, deflect, or in some manner counteract attempts at unauthorized use of information systems[20]. Generally it consists of a computer, data, or a network site that appears to be part of a network, but is actually isolated, (un)protected, and monitored, and which seems to contain information or a resource of value to attackers.

2.5 Android Security

Android runs on top of the Linux 2.6 kernel[21], therefore Android Linux kernel handles security management for the operating system.

2.5.1 Android Security Architecture

Privilege Separation

Android operating system requires every application to run with its own user identifier (uid) and group identifier (gid). The philosophy behind this design is to ensure that no application can read or write to code or data of other applications, the device user, or the operating system itself[21]. This feature is also known as *sandboxing*.

Application Code Signing

Any application that is to run on the Android operating system must be signed[22]. Android uses the certificate of individual developers in order to identify them and establish trust relationships. The operating system will not allow an unsigned application to execute. Although, the use of a certification authority to sign the certificate is not required, and Android will happily run any application that has been signed with a self-signed certificate.

Permission

For an Android application to work correctly, the developer has to make sure to add request for appropriate permission in the applications *AndroidManifest.xml* (list 4.8). This allows the application to request permission to use the system component that handles the specific task. The permission will need to be granted at install time. When the user installs an application, the user is presented with a list of permissions that the application requests. The user cannot selectively allow of disallow individual permission[23]. The user is prompted with the screen similar to figure 2.5.

```
<uses-permission android:name="android.permission.RECORD_AUDIO" />
```

Listing 2.4: Part of permission request declared in AndroidManifest.xml

2.5.2 Android's Bouncer Service

Bouncer[24] is a service from Google which provides automated scanning of Android Market for potentially malicious software without disrupting the user experience of Android Market or requiring developers to go through an application approval process. The service performs a set of analyses on new applications, applications already in Android Market, and developer accounts. Once an application is uploaded, the service immediately starts analyzing it for known malware, spyware and trojans. It also looks for behaviors that indicate an application might be misbehaving, and compares it against previously analyzed apps to detect possible red flags.

2.5.3 Related Work

There are previous works on developing malware detection tool. These are some of the notable ones.

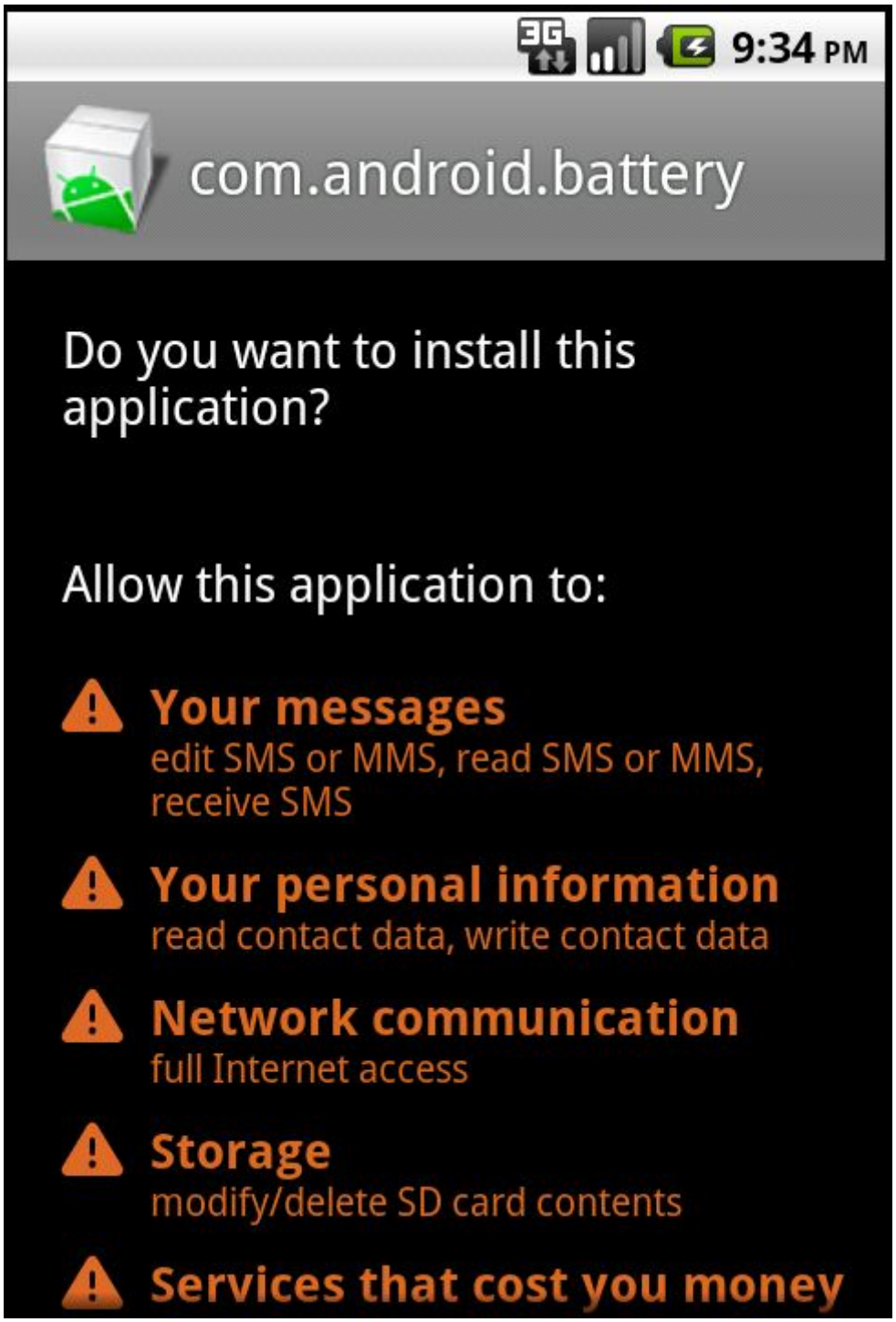


Figure 2.5: Permission request screen

MADAM

MADAM (Multi-Level Anomaly Detector for Android Malware)[25] uses 13 features to detect android malware for both kernal level and user level. MADAM has been tested on real malware found in the wild and uses a global-monitoring approach that is able to detect malware contained in unknown applications (not previously classified).

Monitoring smartphones for anomaly detection

Schmidt et al.[26] monitors smartphones to extract features that can be used in a machine learning algorithm to detect anomalies. The framework includes a monitoring client, a Remote Anomaly Detection System (RADS) and a visualization component. RADS is a web service that receives, from the monitoring client, the monitored features and exploits this information, stored in a database, to implement a machine learning algorithm.

pBMDS

Xie et al.[27] proposes a behavior-based malware detection system (pBMDS) that correlates user's inputs with system calls to detect anomalous activities related to SMS/MMS sending.

Kirin Security Service for Android

Enck et al.[28] and Ongtang et al.[29] propose Kirin security service for Android, which performs lightweight certification of applications to mitigate malware at install time. Kirin certification uses security rules that match undesirable properties in security configuration bundled with applications.

TaintDroid

Enck et al.[30] introduce TaintDroid. TaintDroid monitors applications in real-time, verifying and "Taintin" data transmitted from the device. When an application executes a native method TaintDroid tags and patches the call, alerting the user of the applications activities. Because the tool is monitoring applications at a lower level, the users device bootloader must be unlocked and new firmware installed, voiding the devices warranty.

2.6 Summary

This chapter presented an overview of Android architecture. It shows how Android operating system is designed. The chapter also how Android OS is different from desktop OS and how Android application is structured. Then the chapter talked about detail of malware and malware defend mechanism. The chapter also talked about the security of Android. It described about how Android defends against malware. Google also has its own tool running behind Google's Play Store to scan for malware[31]. Finally, the chapter presented some of the previous work related to the field.

Chapter 3

Design

This chapter discusses how we design our system to analyse and classify smartphone malware by using code analysis and machine learning technique. In our design, an android application is first decompiled with APK tool. The decompiled code (smali) is examined and a hook is inserted to record activities of the application. Python script is run to scan through the user's permission requested by the application. The data gathered altogether is combined to create a vector for machine learning to classify the application as benign or malicious.

The structure of this chapter is as follows: Section 3.1 describes the motivation behind the research. Section 3.2 gives the requirement of the system. Section 3.3 describes decision made to the design of the system. Section 3.4 shows the architecture of the system.

3.1 Motivation

Among the various mobile operating systems today, Android has experienced more attacks since it is an open source operating system. A system that can classify an unknown Android application can benefit users and prevent users to install malware on their smartphones. The approach in this research is different from antivirus technique where it relies on de-

tecting malware based on unique signatures. Although it is very precise, it is of no value against unknown threats and it requires constant signature updates[32]. Anomaly-based approaches, on the other hand, depend on classifiers to train a system to differentiate between normal and malware behaviour, which can be used to detect anomalies so as to discover unknown malwares. Although, the machine learning classifiers has proven to provide more detection accuracy rate[33], this technique presents a main challenge: we must extract some sort of feature representation of the application[34] without having the application's source code. Although, there had been existing research with the use of machine learning to detect Android malware [35],[36],[37],[34],[38],[39], this research is aiming to achieve a better result in term of accuracy by including a wide range of static and dynamic features extracted from Android APK.

3.2 Requirements

Before we design our system, we have outlined a set of requirements for the system to follow.

R1 Extract runtime features from applications.

R2 Extract permissions requested by the applications.

R3 No access should be required to the applications source code. All the required is access to the applications APK file.

R4 Train a machine learning system by using a set features and permission from R1 and R2.

R5 The system is able to identify Android application to be malware or benign app.

R6 Use a classifier to identify any given Android application as benign or malicious.

3.3 Design Decisions

Five key design decisions are discussed below.

3.3.1 Extracting Runtime Features

To extract runtime features we execute the application in an emulated Android virtual machine. We rejected using a modified Android emulator because this is not a portable approach and we wanted to be able to use our system with real physical phones in the future. We experimented with source code manipulation using AspectJ, and although successful, this doesn't meet the requirement R4. Therefore, we focused on hooking API calls at the byte code level. To intercept Android API calls, our approach is to build instrumentation framework to instrument the APK. An instrumentation framework provides the tools needed for monitoring arguments a method takes and the return value. Hook method stores a methods arguments that we will use these information to create application vector. The framework will invoke the appropriate methods during the real time execution of the APK. This framework does not require any help from the user and the vector that stores the information will be generated automatically.

1. Create a trace of Android API calls.

2. Experimented with AspectJ to add hook to compile-time, required changing ant script.

3. No access to source code, adding hook to byte code.

4. Unpack APK, extract classes, decompile in assembly language, insert hooks that call a monitoring library

5. Compile back, repack APK, sign APK

6. Adding monitoring, add new local variables, need to be able to allocate new unused Dalvik registers.

```
.class public Lcom/example/helloworld/MainActivity;
.super Landroid/app/Activity;
.source "MainActivity.java"


# direct methods
.method public constructor <init>()V
    .locals 0

    .prologue
    .line 6
    invoke-direct {p0}, Landroid/app/Activity;-><init>()V

    return-void
.end method


# virtual methods
.method public onCreate(Landroid/os/Bundle;)V
    .locals 1
    .parameter "savedInstanceState"

    .prologue
    .line 10
    invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V

    .line 11
    const/high16 v0, 0x7f03

    invoke-virtual {p0, v0}, Lcom/example/helloworld/MainActivity;->setContentView(I)V

    .line 12
    return-void
.end method
```

Listing 3.1: Example of a .smali file

3.3.2 Modularising the Instrumentation Code

The file monitor.smali is generated at an earlier time by creating a separate Android project containing only this class. This project is then compiled, the classes.dex file extracted and baksmali used to extract the assembly language monitor.smali. With this approach, the Android project is written in JAVA which makes it easy to create instead of having to create it in byte code. This is hooking class can be reused many times with new target applications without requiring it to be rewritten. The hooks to

call the Instrumentation class needs to be inserted into the dis-assembled classes.dex. This procedure will produce a modified file that contains the hooks

3.3.3 Hooking to Intercept Intents

Android API also invoked using events. Some events can be intercepted by adding global broadcast receivers but not all and order of installation will affect success. We observed that to create an intent actually requires a method call. Our approach is to intercept events by hooking method calls that create the intents.

3.3.4 Static Analysis

We could use the hooking mechanism to tract permission request at runtime, although we are interested in what permission are granted regardless of whether they are used or not. Therefore, we extract these information from the *AndroidManifest.xml*[40].

3.3.5 Format of Output

We decided to use WEKA to perform the machine learning, therefore, we adopt the standard ARFF file format used by WEKA.

3.4 System Architecture

The instrumentation system is designed to intercept any API calls and record the activities of the application by using an instrumented class. Figure 3.1 present the architecture of the instrumented system.

The instrumented class *monitor.smali* file will be included with the dissembled APK to form a new modified APK. This will be explained further below.

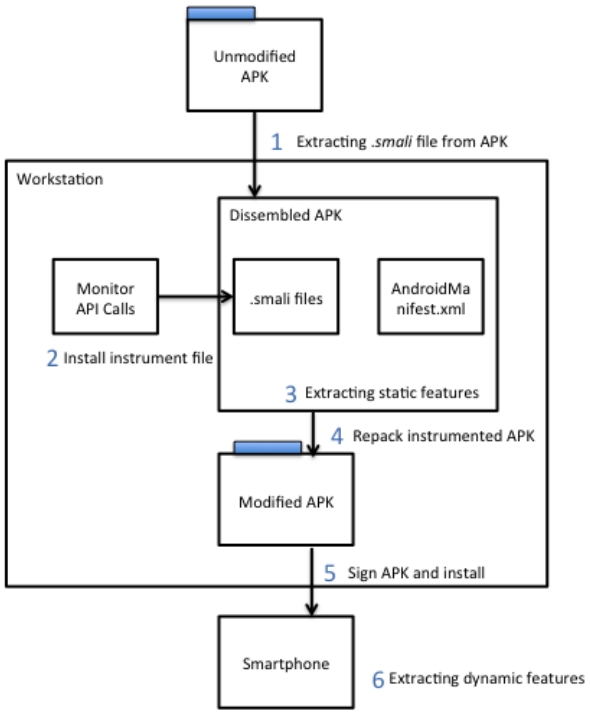


Figure 3.1: System Architecture

The file *monitor.smali* is generated at an earlier time by creating a separate Android project containing only this class. This separate project is compiled into an APK and then the APKtool is used to extracted the assembly language *monitor.smali*. This hooking class can be reused many times with new target applications without requiring it to be rewritten.

First, the APK needs to disassembled into a Smali code, that will be done be extracting the classes.dex file from the target APK. After that, the Baksmali tool must be used to dis-assemble the class of the APK from a Dalvik byte code into a Smali code. The hooks to call the Instrumentation class needs to be inserted into the dis-assembled classes.dex. This procedure will produce a modified file that contains the hooks. In addition to this modified file, the instrumentation.smali file has to be added into the same directory. The files in this directory are assembled using the Smali tool to create a classes.dex file. This in turn is inserted back into the original APK. The modified APK is then signed with Jarsigner using a key generated by Keytool[41].

3.5 Hooking at Byte Code Level

Our approach is to use an instrumentation framework to instrument the APK. An instrumentation framework provides the tools needed for monitoring arguments a method takes and the store those values for later use. Our system has benefitted from such a framework by initializing an array of objects; the before hook method stores a methods arguments, and a variable to store the return value for the after hook method. The framework will invoke the appropriate methods during the real time execution of the APK. The before method take an array of objects that are the arguments of a method and then iterates through them and keep them in a file. The after method takes the return value as an argument and store it before finishing the execution of the method. This framework does not require any help from the user and the array that stores the arguments will be generated

automatically.

This section describes the design of the parsing and instrumentation algorithms of our system.

3.5.1 Parsing

A target class needs to be parsed to locate the implementation of the class. After displaying the methods the user will have the chance to decide which method is to be instrumented. In Smali/Baksmali the code starts each method with the word .method, as listing 4.4 shows; this helped to determine the start of a method.

```
# direct methods
.method public constructor <init>()V
...
...
.end method
```

Listing 3.2: Snippet of a .smali code

The information of the parsed class will be stored into an ArrayList of string to be used later on. The ArrayList is used because it is easy to be extended dynamically.

```
Input: The Smali/Baksmali code of the target class
while input file still has data do
        if the first line of the input file has the word ".method" then
                print out that line
        end
add that line to an ArrayList
get the next line
end
```

Listing 3.3: Parsing Algorithm

3.5.2 Parsing the Instrumentation Class

Disassembled classes of the target APK start with some information that indicate the name of the class, the file path, inheritance if there is any and the original file name, as listing 3.4 shows.

```
# class name, also determines file path when dumped
.class public Lcom/packageName/example;
# inherits from Object (could be activity, view, etc.)
.super Ljava/lang/Object;
# original java file name
.source "example.java"
```

Listing 3.4: Parsing Algorithm

Since we will add a new class (instrumentation class) into the APK, the added class has to have the same information that other classes have such as the file path. The following algorithm has been implemented to modify the instrumentation class information.

```
Input: The instrumentation class
while Input file still has data do
        if the first line of the input starts with ".class" then
                replace the information of the instrumentation class to the
                    target class, and keep the name of the instrumentation class
                write into a file
        else
                write into a file
        end
        get the next line
end
```

Listing 3.5: Modifying the instrumentation class path.

The system generates a new file contain the instrumentation class after modifying the file path. Algorithm in listing 3.5 takes the instrumentation class as an argument, and then it looks for the word .class. If it finds it, it will replace the information to be similar to the target class and it will write it into a file; if it does not find it, it will just write the data into that file.

3.5.3 Instrumenting Algorithms

The target class will be parsed and stored into an array of string by applying Algorithm 1. Another algorithm is required to find the method that a user specified to add hooks into it. Since there are two methods that need

to be inserted into the target method, it is important to determine the beginning and the end of the target method. The target method could have more than one argument. Therefore, it is more efficient to store all of the arguments into an array and the pass them to the instrumentation method if they are more than one rather than pass one argument at time.

```
Input: The array of strings that contains the target class
Get the name of the target method from the user
while the array still has information && user input does not equal EXIT do
        get the first value of the array
                if the first value of the array equals to the target method then
                get the number of parameters in the target method
                        if number of parameters > 0 then
                        initialize an array of objects in Smali language
                                foreach parameter do
                                        initialize a space for the parameter
                                        put the parameter into the array
                                end
                                foreach line of the specified method do
                                        get the first line
                                                if the line equals to return then
                                                        get the return variable
                                                else
                                                        return
                                                end
                                                if the line equals to .end method then
                                                        if the method return type is String then
                                                        insert a hook and pass the return value as a
                                                              parameter
                                                        elseif the method return type is an Integer then
                                                        invoke-static method to get the value of the integer
                                                        move the result into a specific register
                                                        insert an after hook and pass the specific register
                                                              as a parameter
                                                        end
                        else // if there is only one parameter
                                pass the parameter to the before method of the instrumentation class
                        end
end
```

Listing 3.6: Algorithm to insert hooks into a specified smali file.

1. Create a trace of which API calls invoke intents.

2. Intercept API calls without the need of original source code.

3.6 Extract Permission

Androids API is controlled by an application permission system. The permission validation mechanism is implemented as part of the trusted sys-

tem process[42]. Each application must declare upfront what permissions it requires, and the user is notified during installation about what permissions it will receive. If a user does not want to grant a permission to an application, he or she can cancel the installation process. The permissions can provide users with control over their privacy and reduce the impact of bugs and vulnerabilities in applications. However, a permission system will be ineffective if developers routinely request more permissions than they require. Overprivileged applications expose users to unnecessary permission warnings and increase the impact of a bug or vulnerability.

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Listing 3.7: A permission requested in AndroidManifest.xml

1. Android permission requests of each application are contained in AndroidManifest.xml file

2. AndroidManifest.xml file is located in main output folder when application dissembled.

3.7 Formatting for Machine Learning

The arguments stored from the instrument class and the requested permission are transform into an application vector which is stored in *.arff* file. This file is the input of WEKA machine learning.

```
@relation android

@attribute ACCESS_COARSE_LOCATION {0,1}
@attribute ACCESS_NETWORK_STATE {0,1}
@attribute BLUETOOTH {0,1}
@attribute BLUETOOTH_ADMIN {0,1}
@attribute CALL_PHONE {0,1}
@attribute CAMERA {0,1}
@attribute INTERNET {0,1}
@attribute READ_CONTACTS {0,1}
@attribute READ_SMS {0,1}
@attribute SEND_SMS {0,1}
@attribute RECORD_AUDIO {0,1}
@attribute READ_PHONE_STATE {0,1}
```

```
@attribute malware {yes,no}

@data
1,0,0,0,1,0,1,0,1,1,0,0,no
0,1,0,0,0,0,1,0,0,0,0,0,no
1,0,0,0,1,0,0,0,1,1,0,0,no
0,0,1,1,0,0,0,0,0,0,0,0,no
1,1,0,0,0,0,1,0,0,0,1,1,yes
0,1,1,1,0,1,1,0,0,0,1,1,yes
0,0,1,1,0,0,1,1,0,1,0,0,yes
0,0,0,0,0,0,1,0,0,0,1,1,yes
```

Listing 3.8: Example of *.arff* file.

3.8 Machine Learning

The classification process has two phases, training and runtime. Training phase is the machine learning task of inferring a function from labeled training data. The training data consist of a set of training examples. A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples in runtime phase. The procedures are as followed:

1. A list of API calls and permissions requested extracted from decompiled application are combined to create a vector needed as an input for machine learning.

2. A training set consisted of known benign and malicious applications is used to train a classifier.

3. Trained classifier is used to classifier android application at runtime.

3.8.1 Training Phase

The following describes the process of the training phase of the system, as shown in Figure 3.2.

1. A set of Android applications are disassembled to obtain *.smali* code and *AndroidManifest.xml* with the help of *APK tool*.

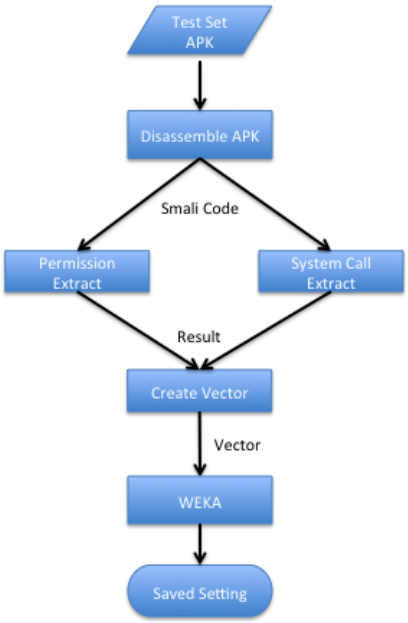


Figure 3.2: Flow chart of training phase

2. A hook is inserted to record and store information of the application.

3. The information of obtained from the previous process is format into an *.arff* file as an input to train the classifier in WEKA.

4. The result from the trainings are saved as a setting to be used to classify the test set.

3.8.2 Runtime Phase

The runtime phase is slightly different from the training phase, as shown in Figure 3.3. The classifier uses the setting from the training phase to classify the application to be whether benign or malicious.

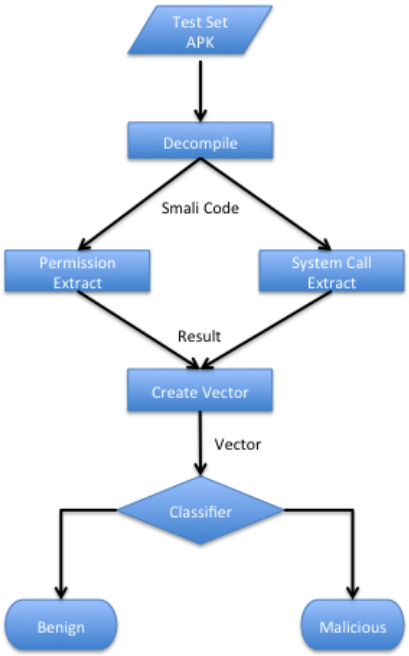


Figure 3.3: Flow chart of runtime phase

Chapter 4

Implementation

This chapter explains the implementation of the system. First, it shows the Instrumentation class that was implemented. it provides an overview of the hookings implementation functions and other tools that have been used to implement the system. The final product contains five classes that are required to parse and instrument the code.

4.1 Tools

Several existing tools were used during the implementation of our system. The first tool is the APKtool[1] which allows to unpack the APK to extract the .DEX file. Second, the Baksmali/Smali[2] tools that were used to disassemble and assemble the APK after the instrumentation. The tool was implemented in Java language using the Eclipse tool. Finally, the Keytool was used to generate a key to sign the APK using the Jarsigner[3] tool.

[1]https://code.google.com/p/android-apktool/
[2]https://code.google.com/p/smali/
[3]http://developer.android.com/tools/publishing/app-signing.html

4.1.1 APKtool

Apktool[4] is a tool for reverse engineering 3rd party, closed, binary Android applications. It can decode resources to nearly original form and rebuild them back after making some modifications. This makes it possible to debug smali code step by step. Also it makes working with android application easier because of project-like files structure and automation of some repetitive tasks like building apk, etc.

4.1.2 Smali/Baksmali

Smali/Baksmali[5] is a Java-based assembler/dis-assembler for the dex format used by Dalvik, Android's VM implementation. The syntax is based on the Jasmin's assembly language syntax and supports the full functionality of the dex format (annotations, debug info, line info, etc.). Smali and Baksmali are Dutch words; the word Smali means assembler and the word Baksmali means dis-assembler.

```
.class public Lcom/packageName/example;
.super Ljava/lang/Object;
.source "example.java"
```

Listing 4.1: A snippet of the beginning of a smali code.

Listing 4.1 shows a typical Smali class header. The first line of the Smali code shows the class path and name. The class path is `Lcom/packageName/` and the class name is `example`. The second line of the Smali code starts with `.super` which represents the class that is `example`. It is inheriting from the `Object` class by default as per the Java specification. The final line of the Smali class header represents the original Java file name.

```
# direct methods
.method public constructor <init>()V
        .registers 1
        .prologue
```

[4]https://code.google.com/p/android-apktool/
[5]https://code.google.com/p/smali/

```
        .line 8
        invoke-direct {p0}, Landroid/app/Activity;-><init>()V
        return-void
.end method
```

Listing 4.2: A snippet of methods in smali syntax.

In listing 4.2, the first line indicates the type of the methods: direct and virtual. Direct methods are the constructors that have extra attributes such as `init` which describes different forms of object initialization. The virtual method is a method that is not static or final, and is not a constructor. The `V` at the end of the method in the second line means the return type of the method is void. The line after defining the method `.register 1` specifies the number of the used registers in a method that need to be executed; the number 1 means it uses only one register. The `.prologue` and `.line` can be mostly ignored, but sometimes line numbers are useful for debugging errors. The next line calls the constructor of our mother class; the `p0` means parameter 0 which is like `this` from Java class. In Smali code the letter `v` represents local registers and `p` represents parameter registers. The line e returns the value `void`. The last line indicates the end of the method.

4.1.3 WEKA

Weka[43] is open source software under the GNU (General Public License). System is developed at University of Waikato in New Zealand. "Weka" stands for the Waikato Environment for Knowledge Analysis. The software is freely available at http://www.cs.waikato.ac.nz/ml/weka. The system is written using object oriented language JAVA. There are several different levels at which Weka can be used. Weka provides implementations of state-of-the-art data mining and machine learning algorithms. Weka contains modules for data preprocessing, classification, clustering and association rule extraction. Figures 4.1 and 4.2 are the screenshots of WEKA software.

Figure 4.1: Weka Explorer

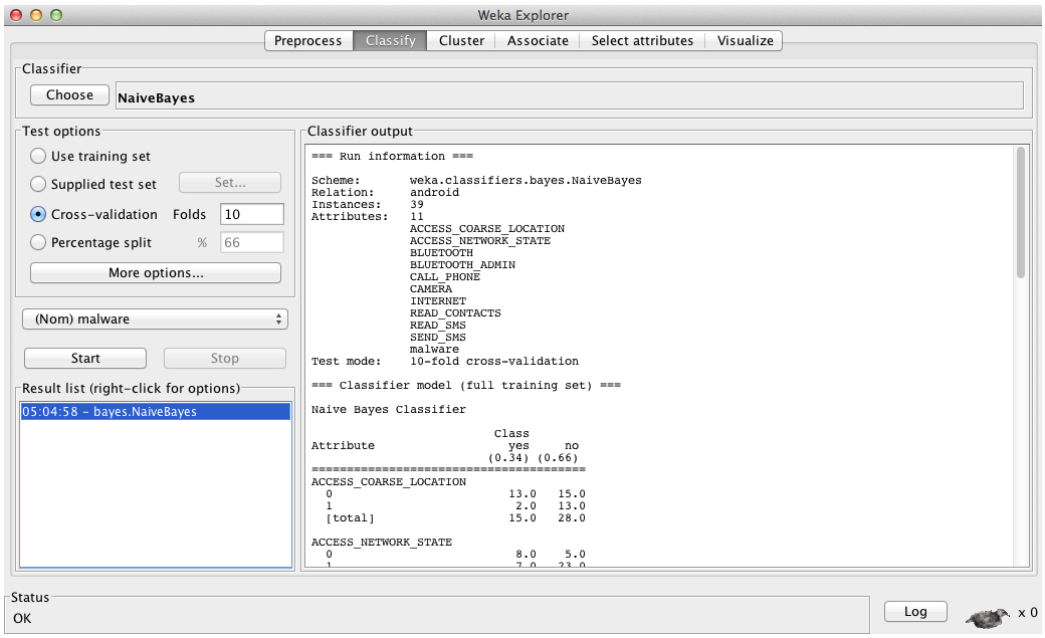


Figure 4.2: Example of Weka result page

4.1.4 Python

Python[6] is an interpreted scripting language and has gained a reputation as being easy to learn. The syntax of the language is designed to be readable. Python has ability to do regular expression. A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern.

```
for line in readfile1:
        if repermission.search(line):
                quotes = re.findall(r'"[^"]*"',line)
                for quote in quotes:
                        outfile1.write(quote)
```

Listing 4.3: A snippet of Python script

4.2 Implementation

The Instrumentation class is implemented as follows. When we instrument method calls, we use the instrumentation class to log the detail of the method calls. The instrumentation class takes an array of objects and store them in a file. This class is instrumented into an Android application by following this procedure.

4.2.1 Adding Hooks

Since hooks will be added based on the number of the parameters to create a space for them in the Array of object that the Instrumentation class takes. If a method has no parameters then it will only add current object of the instrumented class. The number of the parameters must be calculated. Therefore, a class called `getRegister()` is implemented to find out the register number. This method will iterate through the method of interest

[6]https://www.python.org/

and calculate the number of the parameters. Usually, the parameters are listed after the beginning of a method, as Listing 4.4 shows.

```
.method public
print(Ljava/lang/Object;Ljava/lang/Object;Ljava/lang/Object
   ;Lj
ava/lang/Object;)[Ljava/lang/Object;
        .registers 8
        .parameter "a1"
        .parameter "b"
        .parameter "c"
        .parameter "d"
...
.end method
```

Listing 4.4: A method that has four parameters.

After getting the number of the parameters, an array of object will be initialized and the value of the parameters will be inserted into the array, as Listing 4.4 shows.

```
array.add(i+getRegisters.countReg+2, "const/16␣v1,␣0xa");
array.add(i+getRegisters.countReg+3, "new-array␣v0,␣v1,
[Ljava/lang/Object;");
array.add(i+getRegisters.countReg+4, ".local␣v0,
a:[Ljava/lang/Object;");
```

Listing 4.5: Initialization of an array of object in Smali language.

The array is now initialized, and a space for each parameter needs to be initialized to be able to add the parameters. The method will keep creating spaces for the parameters so long as there are parameters.

```
int inc = 1;
int nx = 2;
for (int i1 =0; i1 <= getRegisters.countReg; i1++){
```

```
array.add(i+getRegisters.countReg+4+inc,"const/4␣v1,␣0x"+i1
   );
array.add(i+getRegisters.countReg+4+nx,"aput-object␣p"+i1+"
   ,␣v0,␣v1");
inc += 2;
nx+= 2;}
array.add(i + getRegisters.countReg + 3 + nx , "invoke-
static␣{p0,␣v0},␣"+path+InstCode.fileName+";-
>"+"before"+"(Landroid/content/Context;[Ljava/lang/Object;)
   V);
```

Listing 4.6: Inserting the parameters into an array and passing the array to the before method.

After that, the value of the parameter will be added (listing 4.6 - line 5). Once this method adds all the existed parameters values into the array, a call to the before method of the instrumentation class will be added and the array of the parameters will be passed onto it.

To read the return value, the method will keep iteration through the target the method, when it finds the word return it will skip it and read the returned value. Reading a returned string value is different than reading an integer value. The integer value requires more implementation. Therefore, this method has a condition to distinguish between integer and string value.

In Smali the letter I indicates an integer and the word string indicates a string. Thus, the condition in this method will look for these signs. If the returned value is a string then the register identification of that value will be passed to the after method, where if it is an integer then it gets its value and stores it into a register and passes that register identification to the after method.

After doing all the modification that is mentioned above the system will generate two Smali files: instrumentation and a copy of the modify target file. The user then needs to insert these files into the original APK.

In addition, the APK must be signed to be able to install it in an Android phone. Signing the APK requires the user to know the name that is used to sign the original APK, then create a signatures name the same as the in the original package to be able to resign the APK again.

The system will generate a files, *out.smali* which is the instrumented version of the target class and *monitor.smali* which is the Instrumentation file after modifying the path. Now, the user only needs to modify the name of the *out.smali* to match the target class name, insert these file into the APK and sign the APK.

4.3 Scripts Used to Implement System

This section shows some of the commands and scripts used throughout this research.

4.3.1 Disassemble

The APKtool command used to disassemble the APK file to obtain *.smali* files and *AndroidManifest.xml*.

```
apktool d -f OriginalAPK.apk
```

Listing 4.7: APKtool command used to dissembled APK

4.3.2 Extract Permissions

We use regular expression operations in Python to search for any permissions requested in AndroidManifest.xml file. We use ”permission” as the keyword in our search.

```
repermission = re.compile('permission', re.M)

for line in readfile:
```

```
        if repermission.search(line):
                quotes = re.findall(r'"[^"]*"',line)
                for quote in quotes:
                        outfile.write(quote)
                        outfile.write("\n")
```

Listing 4.8: Part of Python script to search and store the requested permissions

4.3.3 ARFF File

To transfer the information obtained from the previous steps into a vector needed for the WEKA[7] (Attribute-Relation File Format or ARFF)[44]. ARFF files have two distinct sections. The first section is the Header information, which is followed the Data information. The Header of the ARFF file contains the name of the relation, a list of the attributes (the columns in the data), and their types. Listing 4.9 shows an part of Python script used to generate ARFF file.

```
#Create the structure of the .arff file
outfile.write("@relation␣android\n\n")
outfile.write("@attribute␣ACCESS_COARSE_LOCATION␣{0,1}\n")
outfile.write("@attribute␣ACCESS_NETWORK_STATE␣{0,1}\n")
outfile.write("@attribute␣BLUETOOTH␣{0,1}\n")
outfile.write("@attribute␣BLUETOOTH_ADMIN␣{0,1}\n")
outfile.write("@attribute␣CALL_PHONE␣{0,1}\n")
outfile.write("@attribute␣CAMERA␣{0,1}\n")
outfile.write("@attribute␣INTERNET␣{0,1}\n")
outfile.write("@attribute␣READ_CONTACTS␣{0,1}\n")
outfile.write("@attribute␣READ_SMS␣{0,1}\n")
outfile.write("@attribute␣SEND_SMS␣{0,1}\n")
outfile.write("@attribute␣malware␣{yes,no}\n\n")
```

[7]http://www.cs.waikato.ac.nz/ml/weka/

```
outfile.write("@data\n")
```

Listing 4.9: Python script to generate ARFF header

The Python script used to generate data of the ARFF file looks like the one in the list 4.10

```
myArray = [accesscoarselocation, accessnetworkstate,
   bluetooth, bluetoothadmin, callphone,
        camera, internet, readcontacts, readsms, sendsms]

def my_range(start, end, step):
    while start <= end:
        yield start
        start += step

for x in my_range(0, len(myArray), 1):
        if x != len(myArray):
                if myArray[x].findall(buffer):
                                outfile.write("1,")
                else:
                        outfile.write("0,")
        else:
                outfile.write("yes\n") #yes for malicious,
                   no for benign
```

Listing 4.10: Python script to generate ARFF header

4.3.4 Reassemble

The APKtool command used to reassemble the instrumented class back together to form an instrumented APK is as follows in the list 4.11.

```
./apktool b app_name
```

Listing 4.11: APKtool command used to reassemble APK

4.3.5 Sign APK

Signing application is critical to installing an APK on a device or emulator, as otherwise the system will refuse to install it. The command executed to sign the APK is as follows in the list 4.12:

```
.jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -
   keystore android.keystore -signedjar APK_name alias_name
```

Listing 4.12: Command used to sign APK

4.3.6 Install APK

To install APK, the adb function is executed using Javas ProcessBuilder class. The command executed is as follows in the list 4.13:

```
adb -s [device_Name] install -r [app_path_signed]
```

Listing 4.13: Command used to install APK

Chapter 5

Experimental Evaluation

This chapter present the results of the experiment. It explains the detail of the Android applications used in this experiment, including the source of the applications and the size of the training set and the test set. Also, the detail classifiers that used to classify Android applications and the discussion of the results from the experiments are presented in this chapter.

5.1 Purpose of Experiments

The purpose of this experiment is to evaluate the accuracy of a binary classifier for Android malware based upon dynamic and static features of the applications. Five different machine learning classifiers are tested in the experiment in order to determine which classifier can give the best result.

5.2 Data Collection

The dataset used in this study were obtained by multiple sources. These included Google Play Store, Android security community, public websites, etc. We downloaded 100 free Android applications from Google

Play Store in New Zealand. The applications were selected from 5 different categories (20 applications from each category) included, communication, entertainment, productivity, music, and games. Since, Google has its own security tool, codenamed *Bouncer*[24], which provides automated scanning of Android Market for potentially malicious software. Therefore, we considered applications downloaded from Google market place as benign applications. The reason why we only use 100 applications is because it considerably takes a amount of time to instrument and extract features from each application. The malicious applications are collected from multiple websites[45, 46] and some researchers even write their own malware[47]. We have downloaded about 600 malicious applications from the websites but ended up using 100 applications with the same reason as of the benign applications. Unlike iOS, Android user can choose to download and install Android application from any other sources other than Google if they want to. Many websites have android applications available for download for free, so we downloaded some of them from those websites. There are many research groups/communities which interested in Android security, we also asked these people to provide us with the malicious applications.

5.2.1 Source of Android Applications

There are many ways to obtain Android applications. Google has its own app store, called "Google Play Store"[1], where people can download Google approved applications. To be in the market place, the application has to pass through Google's system of verifying the application in a variety of aspect.

[1]https://play.google.com/store

5.2.2 Training Set

The classifiers used in WEKA are trained by a 50 of each benign and malicious application vectors, 100 vectors in total. The training dataset was randomly selected from the pool of benign and malicious. The data was randomly arranged in the dataset using a pre-processing feature in WEKA, which is a machine learning software, and in particular a package called weka.filters.unsupervised.instance.randomize. This way, the classifier training process is more accurate.

5.2.3 Test Set

We use the remainder of the data from the pool to be our test samples. We use the data of only 100 applications, the same size of data set (100 vectors in total) in the test phase, due to the constraints in the data preparing process since we have not implement the process to instrument the application in bulk. The application is run one-by-one in order to generate a vector. This size is comparable to the experiment in [48].

5.3 Classifiers

The classifiers work with a labelled dataset and find a pattern to build a proper detection mode. In this experiment, 5 classifiers have been used, ranging from a simple to more complex and powerful ones. We also want to compare our result with the other researcher, Su et al.[49]. The 5 classifiers are as follows:

- Naïve Bayes: It is a simple probabilistic classifier based on the Bayes theorem with a strong features independence assumption, meaning that, it presumes there is no dependency between various dataset features, something that is rarely true.

- K-Nearest Neighbor: It is one of the most straightforward classifiers, also referred to as KNN. Regardless of its simplicity, it has accomplished a number of pattern recognition tasks. In this experiment, 3 neighbours were chosen to perform this classifier, this is the stand

- Decision Tree (J48): It is a renowned, relatively simple classifier. It is an open source Java implementation of the C4.5 decision tree. The model looks like a tree and a decision is made based on whether a record of data belongs to a branch or not. It is a popular classifier since it is easy to interpret and explain.

- Multi-Layer Perceptron (MLP): The multi-layer perceptron (MLP) is a type of artificial neural network (ANN) consisting of a network of neuron layers inspired by the human brain. It has been widely employed among researchers in various fields such as banking, defence, and electronics. MLP has medium-level complexity.

- Support Vector Machine (SVM): The support vector machine was developed in the reverse order of a neural network. It has a robust theoretical background, which renders it superior in terms of performance compared to neural networks. However, it is complex, hard to interpret, CPU-bound, and memory-intensive.

5.4 Validation

To evaluate each classifier, 2 validation methods known as *k*-fold cross-validation and 70% split were used. The *k*-fold cross-validation[50] method is a means of enhancing the holdout method. The holdout method is one kind of cross validation where the data is divided into two sets, training set and testing set[50]. The function approximator is computed with the training set only. Then the function approximator is asked to predict the output values for the data in the testing set which it has never seen these

output values before. The *K*-fold cross-validation, however, the data set is divided into *k* subsets, then one of the *k* subset will be chosen to serves as the test set. The training set will utilise the remaining *k*-1 subsets which the data are compiled together to form a whole training set. The process is repeated *k* times, then the average error across all *k* trials is computed. The advantage of this method is that it matters less how the data gets divided. Every data point gets to be in a test set exactly once, and in a training set *k*-1 times. The variance of the resulting estimate is reduced as k increases. The disadvantage of this method is that the training phase has to be rerun from scratch k times, meaning it takes *k* times as many computations to perform an evaluation. Specifically, a 10-fold option was used, which is described as applying the classifier to data 10 times and every time with a 90-10 configuration, i.e. 90% of data for training and 10% for testing. The final model is the average of all 10 iterations.

The second method is 70% split, which is defined as using 70% of a dataset for training purposes. The benefit of this method is that it takes much less time compared to the 10-fold method since the process is done once, whereas the same process is done 10 times for the other method. Over-fitting is a drawback of the 70% method, and it occurs when a classifier memorizes a dataset instead of getting trained. Generally, in the majority experiments, like the experiment from [48], the 10-fold method produces better results than the split method which we also observed this in our experiment.

5.5 System Environment

This experiment was throughly performed on a single laptop, 13 inch Macbook Pro 2009. The machine specifications are:

- Processor: Intel core 2 duo 2.26 GHZ.
- Memory: 4GB of RAM.

- Operating System: OS X 10.8. 32-bit.
- Machine Learning: WEKA.
- Java Runtime Environment (JRE): Version 1.7.0_21.

5.6 Result

The aim of the experiment is to evaluate the performance of our technique by using the information from the APK file, both API calls and permissions, to build up a malware detection method. Five different machine learning classifiers included Naïve Bayes, K-Nearest Neighbours (KNN), Decision Tree (J48), Multi-Layer Perceptron (MLP), and Support Vector Machine (SVM) were used to evaluate the performance in this experiment. The result are shown in term of true positive rate (TPR), false positive rate (FPR), receiver operating characteristic (ROC), and area under the curve (AUC).

5.6.1 Accuracy

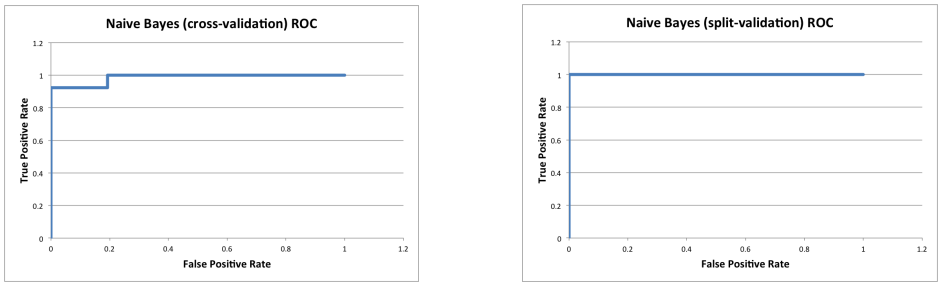
The results are expressed in terms of performance measurements. Detection rate, also known as a true positive rate (TPR), is the probability of correctly detecting an instance as malware. Additionally, false positive rate (FPR) is another measurement defined as the false detection of benign application as malicious. In general, the higher number of the TPR, the better the result is. Conversely, the lower number of the FPR signify the better of the result. There is a tradeoff between these two number, as we tried to maximise our TPR, the FPR will also increase.

5.6.2 Receiver Operating Characteristic (ROC)

An ROC curve is normally used to measure intrusion detection performance. It indicates how the detection rate changes, as the internal thresh-

Table 5.1: The Experimental Results

	10-fold validation		70% split validation	
	TPR	FPR	TPR	FPR
Naïve Bayes	94.90%	10.30%	91.70%	11.70%
K-nearest neighbor (KNN)	84.60%	26.90%	83.30%	23.30%
Decision Tree (J48)	76.90%	30.80%	83.30%	17.60%
MLP	94.90%	6.40%	83.30%	17.60%
SVM	84.60%	23.10%	83.30%	23.30%

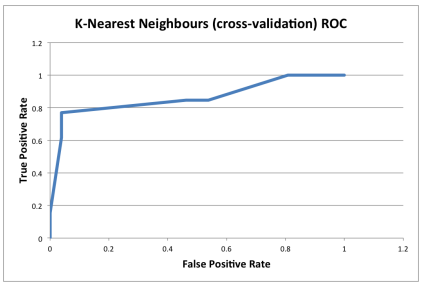


(a) (b)

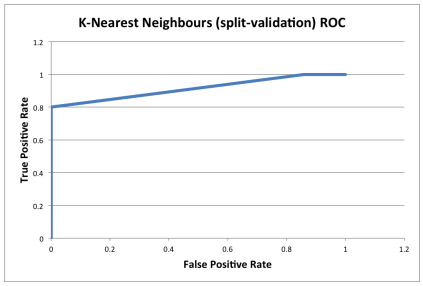
Figure 5.1: Naive Bayes ROC Curve

old is varied to generate more or fewer false alarms. It plots intrusion detection accuracy against false positive probability.

ROC curves signify the tradeoff between false positive and true positive rates, which means that any increase in the true positive rate is accompanied by a decrease in the false positive rate. Then, as shown by the ROC curves, the K-Nearest Neighbours classifier performed the best result. The line in the K-Nearest Neighbours diagram is the closest to the left-hand border and the top border compared to other diagrams, indicated that it offers the finest result among the other classifiers.

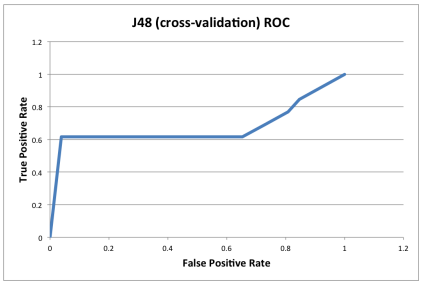


(a)

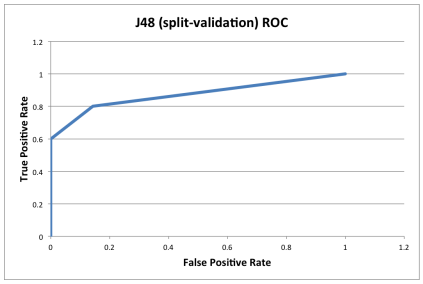


(b)

Figure 5.2: K-Nearest Neighbours ROC Curve

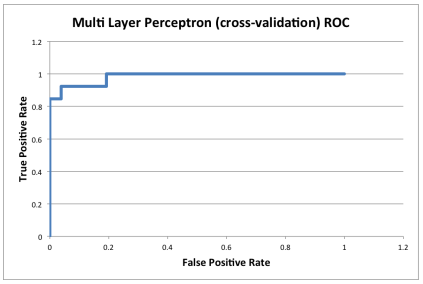


(a)

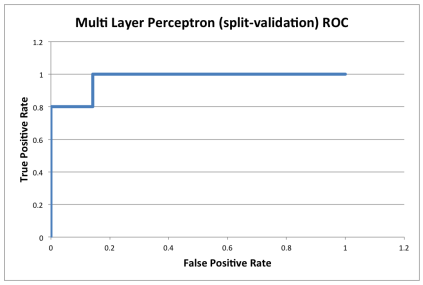


(b)

Figure 5.3: Decision Tree (J48) ROC Curve

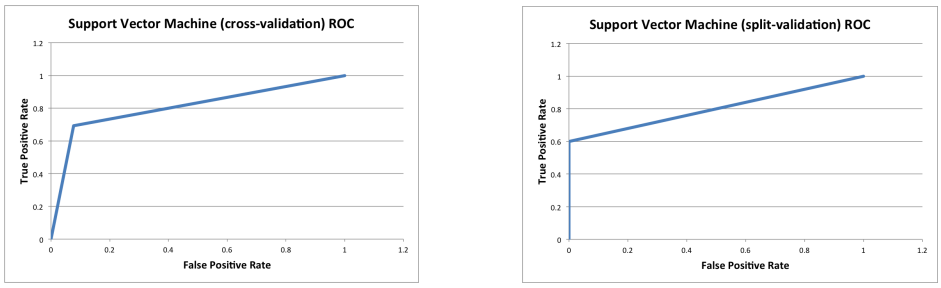


(a)



(b)

Figure 5.4: Multi-Layer Perceptron ROC Curve



(a) (b)

Figure 5.5: Support Vector Machine ROC Curve

5.6.3 AUC

Area under the curve (AUC) is used to measure the accuracy. An area of 1 means a perfect result while a 0.5 value is a worthless result. The AUC point system is as follows: 0.90 1.00 = excellent (A), 0.80 - 0.90 = good (B), 0.70 - 0.80 = fair (C), 0.60 - 0.70 = poor (D) and 0.50 - 0.60 = fail (F).

Table 5.2: Area Under the Curve

	Area Under the Curve	
	Cross-Validation	Split-Validation
Naïve Bayes	0.985	1
K-Nearest Neighbours (KNN)	0.862	0.914
Decision Tree (J48)	0.670	0.871
MLP	0.982	0.971
SVM	0.808	0.800

5.7 Discussion

The result from the experiment shown in previous sections suggests that our detection method can identify Android malware with the accuracy

rate of up to 94.90%. The classifier with the best overall performance is Naïve Bayes, then MLP, KNN, SVM, J48 respectively. Two different types of classify method, 10-fold cross-validation and 70% split-validation, were tested with all five classifiers in the experiment. The result has shown that the cross-validation performs better based on true positive rate. Due to the time constraint, this experiment was only performed on 100 samples, we suggest that if more samples were used in the experiment, the result should yield higher accuracy. The result from this experiment is comparable to Su et al.[49]. Their work utilised network traffic monitoring agent which runs on the actual phone. The agent then send the data to a remote server to do the analysis. Their work produced the true positive rate of 90.80% for decision tree (J48) and 96.7% for random forest classifiers. As the table 5.3 shows the mix result in comparison. Our result from Naïve Bayes is higher than the result from decision tree(J48) of Su et al. but lower than the random forest. The higher accuracy does not necessarily mean that their classifier is better because they focus on a narrower malware domain (botnets). It does suggest though that it would be worth investigate including the dynamic features with ours.

Table 5.3: Result Comparison with Similar Study

	Su et al.[49]	Current Study	
		Cross-Validation	Split-Validation
Naïve Bayes	-	94.90%	91.70%
K-Nearest Neighbours (KNN)	-	84.60%	83.30%
Decision Tree (J48)	90.80%	76.90%	83.30%
Random Forest	96.70%	-	-
MLP	-	94.90%	83.30%
SVM	-	84.60%	83.30%

Chapter 6

Conclusion

The major purpose of this research is to create a system that can identify potential Android malware. In this research we used features extracted from Android application (.apk) files. The extracted data is used as features during a classification process of the applications. Our system is able to do the following:

1. Unpack (disassemble), pack (assemble) an APK file, and sign Android applications.

2. Read the disassembled files and generate information vector of the applications.

3. Generate an instrumented copy of the target class.

4. Identify a potential Android malware.

We performed an evaluation using a collection comprising of 100 benign application and 100 malware from many sources. The results show that Naïve Bayes give an accuracy level of 0.918 with a 0.103 FPR. The high FPR is possible due to the initial classification of the applications. For example, many entertainment applications which are tagged as tools are more similar semantically to the games class. Features, extracted statically

from .apk files, coupled with Machine Learning classification can provide good indication about the nature of an .apk file without running it, and may assist in the detection of malicious applications. The most important features for the detection are extracted using our .dex file parser that can transform contents of the .dex file into standard features (e.g., strings, types, classes, methods, fields, static values, inheritance, opcodes).

6.1 Contribution

In this thesis, I have developed a system that can classify an unknown Android application without the need of the source code of the application. Features and permissions were extracted from .apk files to form a vector, and combined with Machine Learning classification, this can provide good a detection system to detect malicious application without the need of running the application. My contribution to the research are as follows:

- Develop a framework for extracting static and dynamic features from a given Android application.
- Development of a binary instrumented mechanism that allows instrumented application to monitoring of calls to the Android API and the use of declared permissions.
- Implemented a system that can identify potential Android malware from APK file.
- Evaluated the functionality and correctness of the system and measured the accuracy of 5 machine learning classifiers.

We achieved an accuracy of 94.90% for the best classifier. This is comparable but lower than the closest related work, but we focused on a broader range of malware.

6.2 Future Work

Through the result has shown that Mobile Honeypot can effectively identify Android malware tool, the system has potential to grow into an even more powerful policy checker however, and in this section we outline areas of future work.

6.2.1 Greater Feature Detection

Currently, only limited features had been integrated to our checker due to the fact that Android malware still available in a limited number. As Android malware keep growing and become more evolved, we should be able to obtain more malware with a variety. I plan to include more features detection to the monitor in order to create a larger vector, more information for the machine learning. The system should then be able to identify Android malware with more accuracy.

6.2.2 Monitoring Physical Phones

I also plan to develop a agent that runs on a physical smartphone as comparable to Su et al.[49] including the network traffic monitoring feature. As the user download an application, the agent would send data to a proxy server to evaluate the application and send back the result to the user. The agent will alert the user for any potential malware can then make a decision whether to uninstall the application. This should provide convenience to the user. The more user, the more application contribute to the project, this should yield a higher accuracy of the classification as a result.

Bibliography

[1] V. Patel, "Tutorial: Java class file format," Jan. 2009. Url: http://www.slideshare.net/aftekandroid/androidresourcemanagerkt, Last accessed 2014-04-24.

[2] M. Piercy, "Embedded devices next on the virus target list," *Electronics Systems and Software*, vol. 2, pp. 42–43, Feb 2005.

[3] R. Llamas, R. Reith, and M. Shirer, "Worldwide smartphone shipments top one billion units for the first time," Jan. 2014. Url: https://www.idc.com/getdoc.jsp?containerId=prUS24645514, Last accessed 2014-04-08.

[4] B. Sanz, I. Santos, X. Ugarte-Pedrero, C. Laorden, J. Nieves, and P. Bringas, "Anomaly detection using string analysis for android malware detection," in *International Joint Conference SOCO13-CISIS13-ICEUTE13* (. Herrero, B. Baruque, F. Klett, A. Abraham, V. Snel, A. C. Carvalho, P. G. Bringas, I. Zelinka, H. Quintin, and E. Corchado, eds.), vol. 239 of *Advances in Intelligent Systems and Computing*, pp. 469–478, Springer International Publishing, 2014.

[5] L. Goasduff and C. Pettey, "Gartner says worldwide sales of mobile phones declined 2 percent in first quarter of 2012; previous year-over-year decline occurred in second quarter of 2009," May 2012. Url: http://www.gartner.com/newsroom/id/2017015, Last accessed 2014-04-30.

[6] T. Lindholm and F. Yellin, *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[7] T. Downing and J. Meyer, *Java Virtual Machine*. O'Reilly Media, 1997.

[8] S. Microsystems, "The class file format," 1999. Url: `http://docs.oracle.com/javase/specs/jvms/se5.0/html/ClassFile.doc.html`, Last accessed 2014-04-30.

[9] D. Ehringer, "The dalvik virtual machine architecture," Mar. 2010. Url: `http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf`, Last accessed 2014-04-29.

[10] G. Paller, "Dalvik opcodes." Url: `http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html`, Last accessed 2014-04-29.

[11] D. Bornstein, "Dalvik vm internals," 2008. Url: `https://sites.google.com/site/io/dalvik-vm-internals.`, Last accessed 2014-04-29.

[12] Google, "App manifest." Url: `http://developer.android.com/guide/topics/manifest/manifest-intro.html`, Last accessed 2014-04-30.

[13] R. Moir, *Defining Malware: FAQ*. Microsoft Corporation, Oct 2003. Url: `http://technet.microsoft.com/en-us/library/dd632948.aspx`, Last accessed 2014-04-29.

[14] Lookout, "Mobile threat report 2011," 2011. Url: `https://www.lookout.com/resources/reports/mobile-threat-report-2011`, Last accessed 2014-04-24.

[15] J. Jamaluddin, N. Zotou, and P. Coulton, "Mobile phone vulnerabilities: a new generation of malware," in *Consumer Electronics, 2004 IEEE International Symposium on*, pp. 199–202, 2004.

[16] H.-S. Chiang and W. Tsaur, "Identifying smartphone malware using data mining technology," in *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, pp. 1–6, 2011.

[17] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy*, pp. 95–109, IEEE, May 2012.

[18] K. Richards, "Network based intrusion detection: A review of technologies," *Computers & Security*, vol. 18, pp. 671–682, Jan 1999.

[19] M. Miettinen and P. Halonen, "Host-based intrusion detection for advanced mobile devices," in *Advanced Information Networking and Applications, 2006. AINA 2006. 20th International Conference on*, vol. 2, pp. 72–76, IEEE Computer Society, 2006.

[20] L. R. Even, "Intrusion detection faq: What is a honeypot?," July 2000. Url: `http://www.sans.org/security-resources/idfaq/honeypot3.php/`, Last accessed 2014-04-08.

[21] S. Gunasekera, *Android Apps Security*, ch. Android Security Architecture, pp. 31–45. Apress, 2012.

[22] Google, "Signing your applications." Url: `http://developer.android.com/tools/publishing/app-signing.html`, Last accessed 2014-04-30.

[23] R. Johnson, Z. Wang, C. Gagnon, and A. Stavrou, "Analysis of android applications' permissions," in *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on*, pp. 45–46, 2012.

[24] H. Lockheimer, "Android and security," Feb. 2012. Url: `http://googlemobile.blogspot.co.nz/2012/02/android-and-security.html`, Last accessed 2014-04-08.

[25] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "Madam: A multi-level anomaly detector for android malware," in *Computer Network Security* (I. Kotenko and V. Skormin, eds.), vol. 7531 of *Lecture Notes in Computer Science*, pp. 240–253, Springer Berlin Heidelberg, 2012.

[26] A.-D. Schmidt, F. Peters, F. Lamour, and S. Albayrak, "Monitoring smartphones for anomaly detection," in *Proceedings of the 1st International Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications*, MOBILWARE '08, (ICST, Brussels, Belgium, Belgium), pp. 40:1–40:6, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[27] L. Xie, X. Zhang, J. Seifert, and S. Zhu, "pbmds: a behavior-based malware detection system for cellphone devices," in *Proceedings of the third ACM conference on Wireless network security - WiSec '10*, p. 37, ACM Press, 2010.

[28] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security - CCS '09*, p. 235, ACM Press, 2009.

[29] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in android," in *2009 Annual Computer Security Applications Conference*, pp. 340–349, IEEE, Dec 2009.

[30] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for real-time privacy monitoring on smartphones.," in *OSDI*, vol. 10, pp. 255–270, 2010.

[31] A. Toor, "Google's 'bouncer' service scans the android market for malware, will judge you at the door," Feb.

2012. Url: http://www.engadget.com/2012/02/02/googles-bouncer-service-scans-the-android-market-for-malware/, Last accessed 2014-04-08.

[32] P. GarcíaTeodoro, J. DíazVerdejo, G. MaciáFernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *Computers & Security*, vol. 28, pp. 18–28, Feb 2009.

[33] A. Feizollah, N. B. Anuar, R. Salleh, F. Amalina, R. R. Ma'arof, and S. Shamshirband, "Study of machine learning classifiers for anomaly-based mobile botnet detection," *Malaysian Journal of Computer Science*, vol. 26, no. 4, 2014.

[34] J. Sahs and L. Khan, "Machine learning approach to android malware detection," in *Intelligence and Security Informatics Conference (EISIC), 2012 European*, pp. 141–147, Aug 2012.

[35] S. Y. Yerima, S. Sezer, and G. McWilliams, "Analysis of bayesian classification-based approaches for android malware detection," *Information Security, IET*, vol. 8, pp. 25–36, Jan 2014.

[36] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik, "A new android malware detection approach using bayesian classification," in *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pp. 121–128, Mar 2013.

[37] A. Shabtai, Y. Fledel, and Y. Elovici, "Automated static code analysis for classifying android applications using machine learning," in *Computational Intelligence and Security (CIS), 2010 International Conference on*, pp. 329–333, Dec 2010.

[38] A. S. Shamili, C. Bauckhage, and T. Alpcan, "Malware detection on mobile devices using distributed machine learning," in *Pattern Recog-*

nition (ICPR), 2010 20th International Conference on, pp. 4348–4351, 2010.

[39] A. Amamra, C. Talhi, J.-M. Robert, and M. Hamiche, "Enhancing smartphone malware detection performance by applying machine learning hybrid classifiers," in *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity* (T.-h. Kim, C. Ramos, H.-k. Kim, A. Kiumi, S. Mohammed, and D. lzak, eds.), vol. 340 of *Communications in Computer and Information Science*, pp. 131–137, Springer Berlin Heidelberg, 2012.

[40] Google, "System permissions," 2013. Url: `http://developer.android.com/guide/topics/security/permissions.html`, Last accessed 2013-05-12.

[41] S. Marathe, "Android resource management," Mar. 2010. Url: `http://www.slideshare.net/aftekandroid/androidresourcemanagerkt`, Last accessed 2014-04-17.

[42] Y. Chen, G. Danezis, V. Shmatikov, A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security - CCS '11*, p. 627, ACM Press, 2011.

[43] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, Nov. 2009.

[44] G. Paynter, L. Trigg, E. Frank, and R. Kirkby, "Attribute-relation file format (arff)," Nov. 2008. Url: `http://www.cs.waikato.ac.nz/ml/weka/arff.html`, Last accessed 2014-04-14.

[45] Mila, "Contagio malware dump," June 2011. Url: `http://contagiodump.blogspot.co.nz/`, Last accessed 2014-04-28.

[46] Mila, "Contagio mobile malware mini dump," July 2011. Url: `http://contagiominidump.blogspot.co.nz/`, Last accessed 2014-04-28.

[47] X. Jiang, A. Bhattacharya, P. Dasgupta, W. Enck, I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices - SPSM '11*, p. 15, ACM Press, 2011.

[48] A. Feizollah, N. B. Anuar, R. Salleh, F. Amalina, R. R. Ma'arof, and S. Shamshirband, "A study of machine learning classifiers for anomaly-based mobile botnet detection," *Malaysian Journal of Computer Science*, vol. 26, no. 4, 2014.

[49] X. Su, M. Chuah, and G. Tan, "Smartphone dual defense protection framework: Detecting malicious applications in android markets," in *2012 8th International Conference on Mobile Ad-hoc and Sensor Networks (MSN)*, pp. 153–160, IEEE, Dec 2012.

[50] J. Schneider, "Cross validation," Feb. 1997. Url: `http://www.cs.cmu.edu/~schneide/tut5/node42.html`, Last accessed 2014-04-08.