University of Piraeus

Postgraduate Program
M.Sc. *Digital Systems Security*

Master's Thesis

Multiple Layer Hybrid Classification for Android Malware Detection

Anyfantakis Konstantinos
MTE 1902

Supervisor
Prof. Christos Xenakis

June, 2021

Abstract

Because of the ever-increasing number of mobile devices running the Android operating system, as well as their widespread use and diverse application capabilities, such devices have become lucrative targets for malicious apps. Despite mitigating attempts, mobile malware has begun to flourish at an alarming rate. Because Android is an open platform that is fast dominating other rival systems in the mobile smart device industry [3], this has become much more prominent. Experts acquire significant insights into the mechanics of malware using powerful static and dynamic analysis, and machine learning is frequently used to discover unknown harmful software. Nevertheless, the Android operating system, as well as malware associated with it, is always changing. As a result, training a machine learning model with obsolete malware may have a detrimental impact on the predicted detection of more recent malware, so one of the side goals of this thesis is introducing the Omnidroid dataset and the usage of AndroPyTool, already covered in other research works. Apart from that, a new wave of Android malware groups has recently developed that have excellent evasive capabilities, making them far harder to identify using traditional approaches. Various malware detection approaches based on static, dynamic, and hybrid analysis have recently been proposed to make Android devices increasingly safe, however with the growing evolution of malware these methods are nowadays ineffective and imprecise. This thesis not only demonstrates how to employ unique parallel classifiers forming stacked ensemble models to identify zero-day Android malware, but it also discusses how this type of models helps improving malware detection using it on both types of features (static features obtained from static analysis and dynamic from dynamic analysis). On top of that, the suggested approach attempts to fuse the results from these two types, being classified on their own, to aggregate attributes from parallel classifiers using as an example a soft-voting ensemble. The final prediction accuracy on the given dataset was found to be around 91%.

Keywords: Android, malware, detection, machine learning, ensemble, parallel

Table of Contents

List of Figures

List of Abbreviations

1.	AOSP,	Android Open Source Project
2.	GPS,	Global Positioning System
3.	UMTS,	Universal Mobile Telecommunications System
4.	GSM,	Global System for Mobile
5.	IPC,	Inter-Process Communication
6.	OOM,	Out of Memory
7.	HAL,	Hardware Abstract Layer
8.	API,	Application Programming Interface
9.	ELF,	Executable and Linking Format
10.	ART,	Android Run Time
11.	VM,	Virtual Machine
12.	JVM,	Java Virtual Machine
13.	SDK,	Software Development Kit
14.	JNI,	Java Native Interface
15.	URI,	Uniform Resource Identifier
16.	UID,	Unique Identifier
17.	GID,	Global Unique Identifier
18.	XML,	Extensible Markup Language
19.	DEX,	Dalvik Executable
20.	JAR,	Java Archive
21.	IMSI,	International Mobile Subscriber Identity
22.	ML,	Machine Learning
23.	SVM,	Support-Vector Machines
24.	JSON,	JavaScript Object Notation
25.	CSV,	Comma-Separated Values
26.	GUI,	Graphical User Interface
27.	IDE,	Integrated Development Environment
28.	CV,	Cross Validation
29.	AUC,	Area Under Curve
30.	ROC,	Receiver Operating Characteristic

1. Introduction

In order to present the motives behind this thesis, this section leads the rest, ensuring adequate context for understanding the underlying problem and the factors that encourage research to provide information and solutions.

At the present, cyber threats are one of the most difficult and challenging problems affecting contemporary society. These activities are an attempt to disrupt, reveal, modify, disable, steal or obtain unauthorized access to something that has value to an organization or another actor, or make unauthorized use of it. Efforts to address these attacks have resulted in enormous costs [8]. Although the forms in which these cyber attacks are presented and perpetrated, vary, a large number of mechanisms and techniques are also deployed to deal with them.

This research focuses on a specific type of attacks, those carried out through executable files containing a malicious payload, also known as malware, software that attempts to obstruct their proper functioning by causing damage to computing devices. Such attacks have a long trajectory, since the first viruses emerged in earlier years. Ever since, they have progressed into various forms, trying to implement different processes like attempting to bypass antivirus and attain the victim, or even using sophisticated methods of obfuscation designed to prevent its tracking. The main problem involving existing malware is broad and complicated. Malware has been found to be a powerful platform that targets a huge number of users and points to critical infrastructures for massive assaults. Carrying this as a starter, when facing such kind of threats, there are two main tasks: to build malware detectors capable of filtering and classifying suspicious samples that integrate malicious pieces of code and, secondly, to mitigate the damage induced when the detection mechanism has been successfully circumvented. As a standalone process in which suspect samples are examined to form an opinion on whether there is malicious or harmless intent, this study focuses on the former issue.

In addition to malware, because of their success, mobile operating systems are everywhere nowadays. This has been found also in major businesses, thereby introducing new strategies and rising the number of telecommuting staff. Therefore, new tools for automated malware detection in mobile devices, particularly those using the Android operating system, need to be investigated, as it accounts for over 80 percent of the market share compared to iOS [12]. Even after the steps widely in use to suppress exploitation among the rising population of Android users around the world, Android malware is at an impressive speed.

The main means of Android app delivery is by app markets, and alongside the official Google Play app store, many unauthorized online app stores are growing. Since these platforms have poor or non-existent safeguards to discourage malicious apps from being uploaded and delivered to the smartphones of consumers, the third-party app stores that have arisen in past years have also been a very prevalent cause of malicious app delivery.

So, attempts to mitigate malware rely on the creation and operation of filters that can reliably determine if a suspect sample can be deemed to be benevolent or malicious. It is possible to determine the variety of actions that the application will perform and to make that decision by extracting a set of behavioral markers. The relevance of this challenge is without doubt, but the vast number of new applications discovered daily render the use of software capable of automatically dealing with large quantities of samples essential. Therefore, analysis must be carried out to review processes that can simplify this mission, preventing malware to target consumers.

Machine learning strategies are increasingly gaining attention as a solution to resolve this issue in this context. They can be used as a tool to launch malware detectors that can accommodate vast quantities of applications and that can categorize malicious or benign software from already labeled samples based on a previous training phase. Machine learning methods, however, involve vast datasets with representative characteristics derived from actual samples, establishing one of the purposes of this study. Therefore, a strategy is suggested for timely identification of Android malware, by concurrent machine learning classifiers that use various algorithms with features that are intrinsically distinct. In the learning stage of the development of the model, a variety of static and dynamic app features are used.

Finally, regarding the structure of this dissertation: First, there is the segment on the theoretical background, where all theoretical knowledge is provided that was deemed fit or beneficial in understanding the purpose for developing an automated tool for Android malware analysis and detection. Next comes the problem statement and approach chapters, where the challenge that this study helps to address, is quickly identified and evidence that is yet to be considered for its resolution are presented. The next part is the chapter of implementation, in which the method of constructing the classifiers suggested is discussed. The outcomes of the implementation are included in the next chapter. This covers aspects of performance assessment of the multiple constructed classifiers. Eventually, the last chapter contains recommended suggestions which could be focused as possible future work.

2. Theoretical Background

This section includes the background of required information concerning Android and its malware realization. This includes a theoretical background about Android as system, Android malware in general, while it also focuses on the basics of using machine learning. It's quite important to note here, that this section tries mostly to offer basic knowledge, in order to better understand the proposed architecture shown later and to further raise awareness of malware's importance.

2.1 Android

Operating systems for new devices are becoming more relevant as smartphones and tablets become more popular. Android Inc., which was acquired by Google and launched as AOSP in 2007, developed a platform based on that principle and through the years, Android has become an operating system that operates mostly on battery-backed low-powered devices and is equipped with hardware such as GPS receivers, cameras, light and direction sensors, WiFi and UMTS networking. It is developed on top of Linux, but modifies it in major ways, including those that violate the popular usability. A reasonable estimation would be that Android and Linux are about 95 percent the same at the kernel level, and about 65 percent or so at the user-mode, but it is difficult to calculate just how far the two OSes vary [14]. This estimation is rendered by taking into account that, apart from a few variations at the kernel level, the majority of the kernel source is unmodified. Such discrepancies are collectively referred to as Androidisms, and most have already been incorporated into the mainline by now [12].

Android is a feature-rich operating system presenting a complete mobile application software stack. Android APIs are a rich collection of device services that are packaged in intuitive class files that allow many useful functions to be quickly accessed. All the materials, frameworks and functionality needed to create an Android application are available are free. The Android app is a smartphone application designed for use on Google's Android platform-powered smartphones which can be written in several different programming languages. While most of them are heavily Java-coded, they rely deeply on a large stack of C++ written native libraries, discussed also in 3.1.1. Developers of software can conveniently navigate the tremendous array of framework resources, software and libraries if necessary in their app.

2.1.1. Architecture

Android as a platform has managed to have a quite simple and yet divided architecture. As seen in figure 3.1, there are 5-6 different software layers, each one serving its own purpose. Starting from bottom to top, due to its open source and free license nature, the Linux kernel offers an excellent Android low-level example. The Linux kernel is the base of the Android platform. The Android Runtime for instance – also seen later in detail, depends on the Linux kernel for underlying functionality such as threading and resource protection at a low level. Using a Linux kernel makes it easier for Android to take advantage of key security features and helps manufacturers of smartphones to create hardware drivers for a well-known kernel. Introducing its own new features in the Linux kernel, Android also adds Binder IPC, the heart of all IPC in Android working like a clock pulse in a circuit, keeping sync by communicating between processes and services and presenting a character device like */dev/binder* which all applications can open. Along with Binder, it adds Anonymous Shared Memory, which is a shared memory system where applications can open a character interface like */dev/ashmem* to construct a region of memory that can then be converted into memory, a Logger, ION Memory Allocator and Low Memory Killer - a layer on top of Linux's own Out-Of-Memory (OOM) killer who, in the event of memory depletion, terminates processes [12].

Immediately after the kernel layer, follows the hardware abstraction layer also known as HAL. Standard interfaces that expose computer hardware functionality to the higher-level Java API platform are provided by it. The HAL consists of several library modules, each of which has an interface, such as a camera or Bluetooth module, for a particular type of hardware item. The Android system loads the library module for that hardware part when a platform API allows a call for hardware access. In other words, a HAL defines a standard specification to be introduced by hardware vendors that allows Android to be agnostic regarding implementations of lower-level drivers used in the previously mentioned kernel [14]. Using a HAL allows to enforce features without impacting or changing the higher level structure.

Many main components and services of the Android framework, such as HAL and other core characteristics, are constructed from native code that includes native libraries written in C and C++. To open the capabilities of some of these native libraries to users, the Android platform offers Java application APIs. For starters, via the Java OpenGL API of the Android platform, you can access OpenGL ES to add support for drawing and manipulating 2D and 3D graphics in your app. Most of the essential portion of Android critical parts, is implemented in C/C++ and compiled into native binaries. Almost all core libraries exist in a filesystem like *system/core* to provide wrappers over kernel features or implement additional functionality like wrapping the ION Memory Allocator socket mechanism and abstracting partition management [14]. User programs are compiled into bytecode, but in the sense of an Android runtime environment as virtual machine, which is an ELF binary where the bytecode runs or compiled ahead of time.

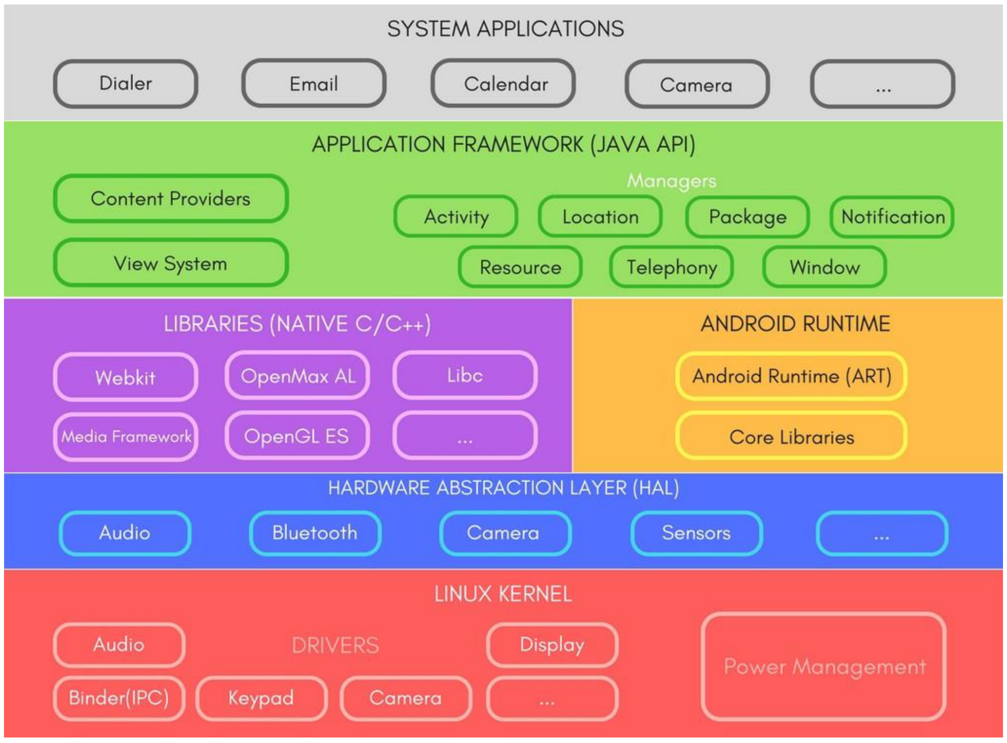


Figure 2.1. Android platform architecture

Now for one of the most important parts of Android, lying together with native and core android libraries, follows the Android Runtime or ART. Android Runtime and its precursor, Dalvik, were primarily developed for the Android project, performing the translation of the application's bytecode into native instructions that are later executed by the device's runtime environment. In general, the Dalvik VM and ART use a register-based architecture that needs less, usually more complex, virtual machine instructions, unlike Java virtual machines, which are stack machines [12]. Using the Android application programming interface, applications are written in Java, compiled into Java bytecodes, and converted to Dalvik or ART instructions as provided. Several Java classes are contained in a single file called .dex. In order to save space, redundant strings and other constants used in different class files are used in the .dex output only once. Also, the Java bytecode is often translated into an alternate instruction set used by the Dalvik VM. ART executes the Dalvik Executable or .dex format and the definition of its bytecode as the runtime. ART and Dalvik are .dex bytecode compatible runtimes, so when running with ART, applications built for Dalvik should operate.

Some techniques that operate on Dalvik, however, do not operate on ART [14]. Some of the features added were ahead-of-time compilation and development / debugging improvements. ART increases the overall performance of execution and decreases power consumption by reducing Dalvik's interpretation and trace-based JIT collection, resulting in increased battery autonomy on mobile devices. At the same time, ART brings quicker program execution, better processes for memory allocation and garbage collection, modern debugging functionality for applications, and more reliable high-level program profiling [12].

Android programs run on a virtual machine, but often they need to break from it, usually to reach unique features on hardware or chipsets. Android Runtime then allows, via the Java Native Interface (JNI), the incorporation of native libraries (ELF shared objects) in the program code. There is no question that manufacturers will be satisfied with pure implementations, as those are limited to the VM and therefore are agnostic to the underlying architecture. In this manner, with no change, Android apps will run uniformly on all chipset architectures. The VM setting, on the other hand, is not without its limitations and disadvantages, such as decompilation. Therefore, it is not at all surprising to see JNI used in performance enhancing applications or finding resistance to reverse engineering [14].

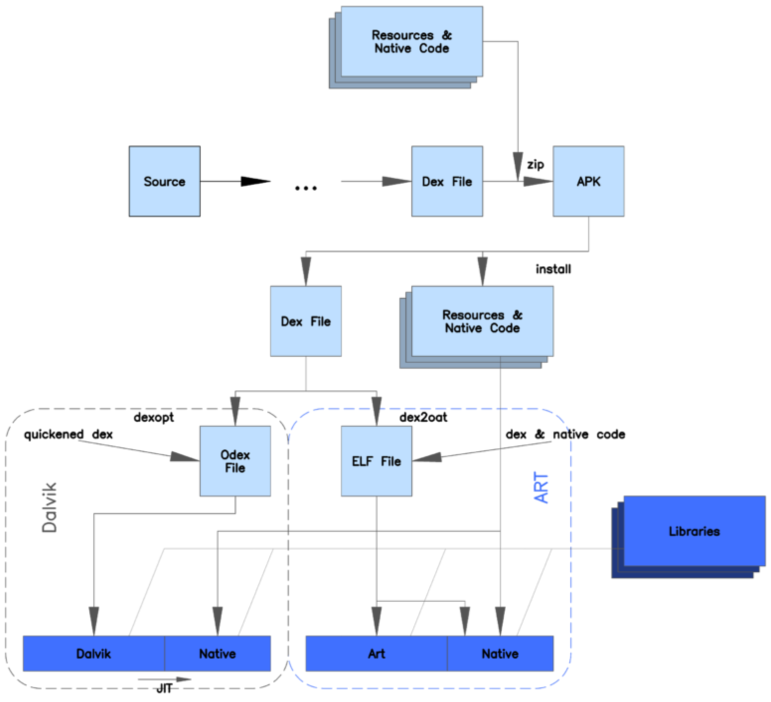


Figure 2.2. Dalvik / ART Architecture

The entire Android OS feature set is also accessible through APIs written in the Java language. By simplifying the reuse of core, modular device components and utilities, these APIs shape the building blocks you need to construct Android apps. It owes its rich collection of frameworks a crucial portion to its popularity because would potentially end up being yet another embedded Linux distribution without them. Android supports the application development process by offering interfaces, encouraging developers to use the higher-level Java language rather than low-level C / C++. The implementation of frameworks further speeds up the process, as developers will rely on the numerous APIs that manage access to graphics, audio and hardware. These are much easier, and run in a far simplified way, unlike X-Windows and GNOME / KDE [14]. Android frameworks are broken into different namespaces according to their features through the use of Java package naming and include a rich and expandable View framework to create the UI of an app, a Resource Manager that provides access to localized strings and layout files, a Notification Manager that enables notifications, an Activity Manager that manages the lifecycle of applications and Service Providers to have access to other apps' data.

Finally, Android comes with a collection of messages, SMS texting, calendars, internet access, contacts, and more key applications. Among the programs the user wishes to use, software included with the framework have no unique classification. So a third-party app can become the default web browser of the user, SMS messenger, or even the default keyboard. The core system apps act both as user apps and feature providers to be accessed by developers from their own software.

2.1.2. Application principles

The most important part of an Android device supported by its own architecture described before, are applications or so called apps. Although at first it was mainly Java that was used to create new apps since the runtimes, frameworks are based on it and JVM but since May 2019 Google declared the Kotlin programming language to be the preferred language for creating new Android apps, which is created to interoperate normally with Java, and its standard library depends on the Java Class Library [12]. As a whole, using Kotlin, Java and C++ languages, new Android apps can be written and be deployed though without any problems.

The Android SDK tools compile the code into an APK, an Android Package, which is an archive file with the .apk extension, along with some data and property files. One APK file includes all the content of an Android app and is the file used to launch the app by Android-powered users. Android Package is the application file format used for the delivery and development of smartphone applications, computer games and middleware using the Android operating system, and a range of other Android-based operating systems. All program code such as .dex files, services, equipment, credentials, and manifest files are stored in an APK file. APK files may have any name needed, as is the case for many file formats, but it might be important for the file name to end with the .apk file extension. Altogether, an APK file usually contains a *lib* directory with compiled code that is platform dependent, a *res* and *assets* directory with resources and app assets used by different content managers, a *META-INF* directory containing a meta-manifest file and other certificates, a classed.dex file that includes the classes compiled by the Dalvik Virtual Machine and the Android Runtime in the dex file format and finally an AndroidManifest.xml file that specifies the program name, version, access rights, referenced library files [16].

Every Android app, covered by the following Android security features, exists in its own security sandbox: The Android operating system is a Linux multi-user system where each software running is a different user. The system assigns a special Linux user ID to all apps by default and sets permissions on all files in an application, so that they can only be reached by the user ID allocated to that application. Each process runs in its own virtual machine, so the code of an app runs independently from other programs and so, each app runs in its own Linux process [34]. When all of the components of the software are to be run, the Android device begins the process and then shuts down the process until it is no longer needed or when the machine needs to restore memory for other applications. The concept of least privilege is also applied by the Android framework. That is, by default, each app only has access to the components it needs to do its job and no more. This provides a very safe environment in which portions of the device, for which approval is not granted, will not be reached by an app. There are, however, forms for an app to exchange information with other software and for an app to access device resources. You can arrange to share the same Linux user ID with two users, in which case they are able to access each other's files. Apps with the same user ID can also arrange to run the same Linux process and share the same VM in order to save machine resources.

There is no single entry point that the device can reach in an Android application like a *main()* function for example, unlike other operating systems. An app component, instead, is a separate point from which the device will independently reach an application and instantiate the component entity. There are various kinds of application components, each serving a function and a lifecycle that determines its creation or destruction [16].

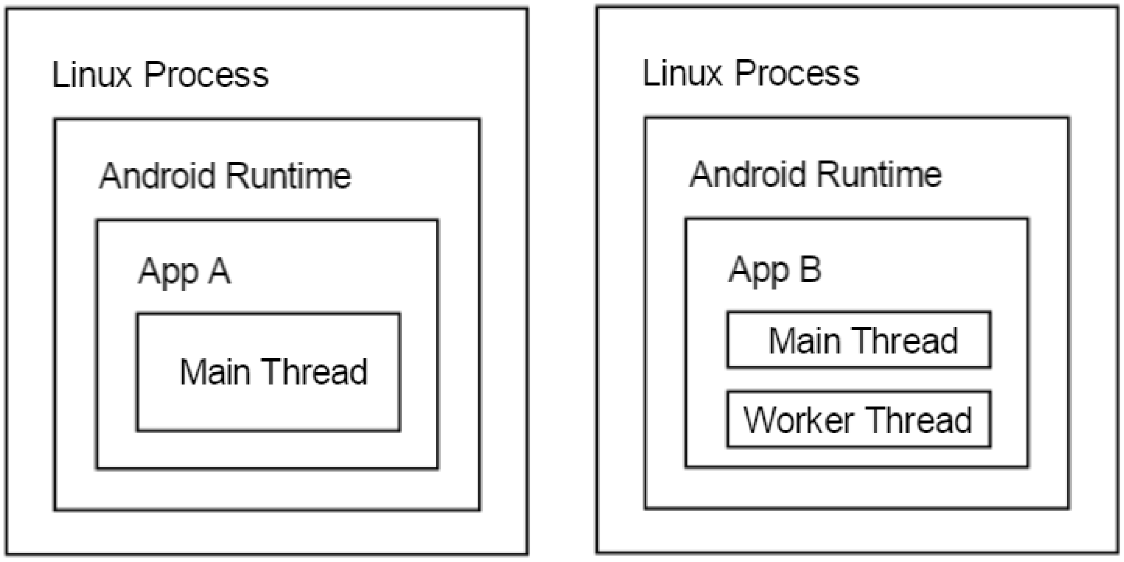


Figure 2.3. Application and process

The first app component that comes in use is the Activity, the point of entry for communicating with the user. It reflects a single user-interface device, which means a single screen with a user interface is represented by an activity. In order to shape a consistent user interface, different activities work together, but each is independent of the others. As such, every one of these activities will start with a separate application. The precise context in which an activity displays users and the number of events in an app depends on how the task is constructed by the developer. Usually, one activity is always defined as the main activity in a multi-activity environment [12]. In order to execute multiple activities, each activity will then initiate another activity. The previous activity is interrupted any time a new one begins, but the device retains them in a stack in order to retrieve them correctly. Normally, the activities implemented as a subclass of the Activity class, know that the formerly used processes require items that can be resorted to by the users like other stopped operations activities, and thereby give more emphasis to holding certain processes intact. They help the app destroy its own process so that the user can return to previous processes and activities in their former state while providing a means for apps to enforce user flows with each other and to organize these flows for the device. A great example for activities is an email app that may have one task that displays a list of new emails, another email composition task, and another for email reading. Then, to encourage the user to post a photo, a camera app will initiate the activity in the email app that composes new mail [12].

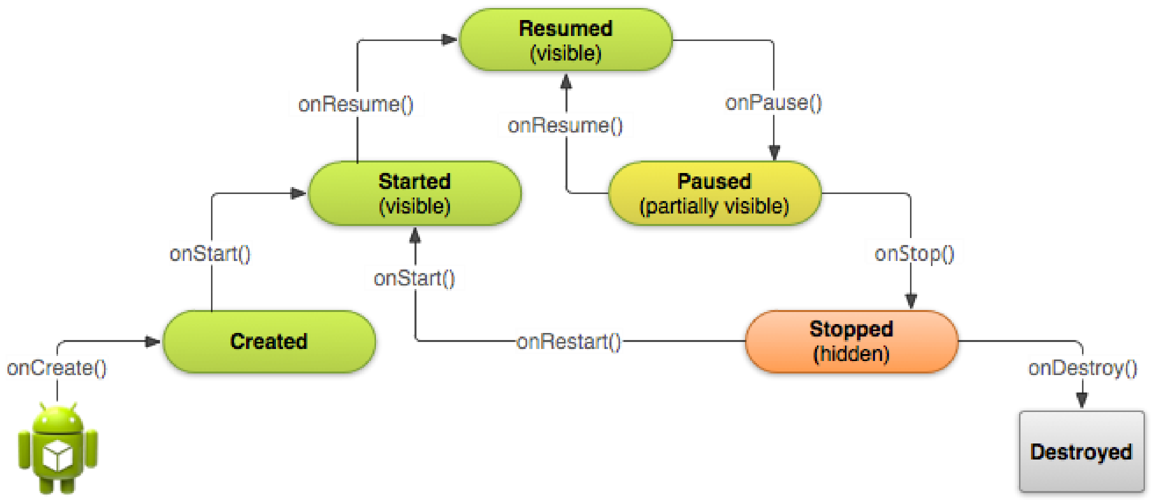


Figure 2.4. Activity lifecycle

Another key component is called Service. It is a general-purpose access point to keep an app going for all types of purposes in the background. It is a part that runs in the background for long-running operations or for remote processes to conduct work. A user interface is not supported in this case. You may connect or attach an activity to a service that is running, while when linked to a service, via the interface exposed, the activity can communicate with the service. Service systems, like all application elements, often run by default on the main thread of an application [12]. There are mainly two kinds of services that tell the system how to manage an app. An app's service that needs to be in the forefront with a message to tell the user about it; in this situation, the machine understands that it should work very hard to keep the activity of the service going, and if it goes out, the user will be disappointed. On the other hand, the user is not explicitly aware that a standard background service is operating, so the device has more flexibility to control it. If it needs RAM for items that are of more urgent importance to the user for example, it will cause it to be destroyed. Live wallpapers, feedback listeners, screen savers, input methods, connectivity facilities, and many other aspects of the core framework [16] are all designed as utilities introduced by apps (as a subclass of Service class) and connected to when they should be running.

Apart from services and activities, there's also another component called Content Provider, which maintains a mutual collection of application data that can be stored in the file system, on the network, in the SQLite database, or some other permanent storage location accessed by the software. Other apps can ask or change the data from the content provider if the content provider requires it. As a subclass of Content Provider class, such a provider specifies the data format it supports and provides a series of methods to allow other applications to query or change the data. A content provider is an entry point in an application for publishing named data objects, defined by a URI scheme, to the system. An app will then specify how to map the data it holds to a URI namespace, transmitting such URIs to other entities who will use them to retrieve the data in turn. An example of content provider in Android, is the one that it provides to manage the user's contact information.

Finally, there are the Broadcast Receivers as part of the app. A broadcast receiver is a part that allows the device to transmit events outside of a normal user flow to the app, enabling the app to respond to broadcast announcements throughout the device. The device will provide broadcasts also to applications that are not currently operating, because broadcast receivers are another well-defined entry into the app. Broadcast receivers are responsible for obtaining the transmission signal and reacting to the information provided by a transmission. For example, a broadcast introduced as a Broadcast Receiver sub-class that announces that the screen has switched off or the battery is down. Although broadcast receivers do not have a user interface, when a broadcast event happens, they will generate a status bar warning to warn the user.

A unique property of the Android device architecture is that any app can initiate any other app's components. If you want the user of the smartphone camera to capture a picture, there's obviously another app that does it and your app will use it to capture it instead of designing a new activity to do it. You don't need to integrate the code from the camera app or even connect to it. Instead, in the camera application that captures a frame, you can simply start the action [16]. As the device launches a component, if it is not already running, it begins the process for that function and instantiates the classes required for the component. For instance, if your software begins a photo-capturing operation in the camera app, that operation will run in the process that belongs to the camera app, not in the process of your app. Thus, Android applications do not have a single entry point, as already mentioned before there is no *main()* function, unlike software on most other platforms. Because each app is run by the device in a different process with file permissions limiting access to other applications, your app is unable to trigger a feature directly from another app. To trigger a component in another app, send a message to the device that defines your *intent* to enable a specific component. The Android system triggers that part for you afterwards. So, an intent, an asynchronous message, aims at runtime to connect individual components to each other regardless of whether the component is part of the same framework. An intent defines the procedure to be done for activities and services, and may determine the URI of the data to be generated annually. For broadcast receivers, the intent specifically determines the transmitting of the broadcast. Intents may not enable the service provider, though. Instead, when targeted by a Content Resolver request, which manages all relevant interactions with the content provider so that the component that's performing transactions with the provider doesn't, the content provider is enabled [12].

All of these app components are declared in the AndroidManifest.xml file which primary function is to inform the device about the app's components. It is located in the root directory of every Android application and actually contains important information about the application and its components with its permissions, packages and used libraries. The device is not noticeable to events, so programs, and content providers that you have in the source but are not announced in the manifest may therefore never run.

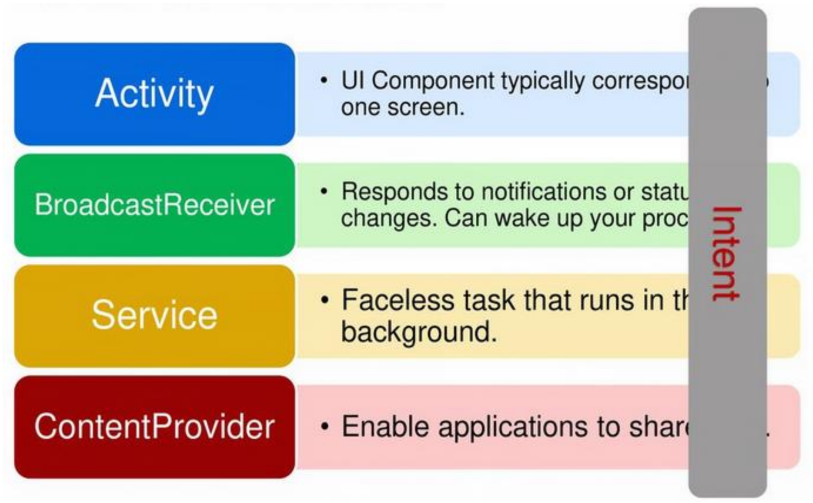

Activity
• UI Component typically correspo one screen.
BroadcastReceiver
• Responds to notifications or statu changes. Can wake up your proc
Service
• Faceless task that runs in th background.
ContentProvider
• Enable applications to shar
Intent

Figure 2.5. App components

2.1.3. Security

Despite major changes for each edition, Android had its security broken a number of times in the past. The flaw was often in Android itself, while at other times in the underlying Linux Kernel. Therefore, it follows that Android security must integrate both environments, Linux and its own, and merge them as successfully and as safely as possible together. On top of the Linux basis, Android builds a rich system, but depends on Linux for nearly all operations at its heart [3]. The Linux provision also allows Android to use the same security mechanisms as the permissions, capabilities, SELinux, and other low-level security protections provided by Linux.

Regarding Linux security model, having remained almost unchanged through the years, it comes up with the most known security features of UNIX family. First of all, a user has a specific user ID called UID, so the real user name does not matter. UID 0 is the most powerful user ID possible, essentially breaks checks and provides access to all files or services regardless of the way authorization checks are done [34]. What follows is the total power of uid over the device. In addition to that, the user has also a numeric default group ID called GID, so the group name does not matter, just like the username, except some GIDs are reserved for device use. File permissions are given to a designated person, group, and "other" that maps read, write, or execute permissions to each one of them. This incredibly narrow model, for which UNIX has been criticized, is accompanied by both files and folders. File access criteria effectively compel the formation of specialist organizations because of their restrictions. Now there are also the SetUID or SetGID binaries, which cause another UID to be presumed or joined during its execution by another party. These unique permissions are immediately given by executing a setID binary. This function is simply a feature used to operate around privileged activities, such as modifying one's UID or password. Finally, it follows that access to system tools such as IPC or UNIX sockets, is an actual verification of user permissions: anything in UNIX can be accessed as files [34]. That is, they can be treated just like files and have the same form of permissions, having the same filesystem interpretation.

Android utilizes the same permission paradigm, which is obtained from the underlying Linux system, but provides a novel, quite different interpretation by saying that the applications are considered as users with IDs, not the actual person using the device, thus creating a kernel-level app sandbox. When an application is installed for the first time, it is given a specific user ID also called app ID by the PackageManager, an object service to obtain different categories of app package-related information currently installed on the device [3]. The same applies for all of the software other than the kernel in Android architecture like native libraries and binaries, app framework or runtime, they all run inside of an global application sandbox system. The root user with UID 0 in Android is limited to as less processes as possible in order to avoid root-owned processes getting exploited. One example of such a process is *init*, the first process to start in Android, responsible for the initial setup of the system [12].

Another security feature implemented in Linux and ported directly in Android, is the capabilities system. If a SetUID binary can be trusted, the model can work in principle. In reality, however, SetUID faces inherent security risks: if a binary SetUID is abused in any way, it may be fooled into compromising the root. Capabilities provide a solution to this issue by dividing root "powers" into different regions, each represented by a bit in a bitmask, and by toggling the bitmask allowing or limiting respectively privileged operations in these regions. For an example, a networking utility like older versions of ping that need to create raw sockets, uses capabilities and with a slice called CAP_NET_RAW allows ping to run as a normal user and decides that only the part that needs the raw socket will be using root [3]. Init also begins most of the core processes of Android as root, and as they launch, these processes have the full bitmask capabilities. However, they lose their privileges until these procedures genuinely do something, and maintain only the capabilities they require.

Furthermore, Android incorporates the notion of isolated services. This function is a type of transference that allows an application to run its services entirely independently with a separate UID in an another process, denying them any other request. As a result, they are unable to scan for device assets and are essentially confined to memory operations. This is very useful for applications such as web browsers [3].

As part of its security model, Android also maintains Security-Enhanced Linux or SELinux and sometimes called SEAndroid. With it, all processes, even those running with root or superuser rights including Linux capabilities, implement mandatory access control. Many enterprises and organizations have contributed to the SELinux implementation of Android. Android can help secure and confine system resources with SELinux, monitor access to application data and server logs or reduce the impact of malicious malware. Generally, SELinux works on the default denial principle, so something not expressly permitted is rejected [34]. It can work in two global modes: the permissive mode, which logs but does not execute permission denials and then enforcing mode, in which denials of authorization are also logged and implemented.

Using enforcing mode, Android provides SELinux and a corresponding security policy that operates through AOSP by default. The key in SELinux is that it is based on labeling. A label assigns to a resource (as an object) a new category, and to a process (subject) a security domain. SELinux can then be enforced such that only processes in the same domain, which is also labeled, can use the resource. For instance, in the *init* domain, no other processes should run other than *init* process which was mentioned earlier.

Now, instead of native code, operating at the level of a virtual machine carries with it immense benefits for controlling activities and imposing stability. The benefit of the ART / Dalvik VM lies therein, that most operations are carried out by means of pre-supplied packages and classes, and those come with authorization controls built-in. The user app is absolutely useless, stripped of all Linux-level functionality and permissions, so all access to the underlying device infrastructure can be blocked right there. In order to perform any operation that has an effect beyond the reach of the program, a process like *SystemServer* (The center of android app system. It initiates all the other system services and registers them with the *ServiceManager*) must be used. Although a call to *SystemServer* may be openly invoked by any application, none has access to its specified permissions, which of course can be reviewed. This check is carried out beyond the phase of the application, because the application has no logical route by which it may, unless it has been delegated to it already, somehow receive such permissions. What follows is that there is no need for unique data structures or specific metadata for the permissions themselves. A permit in Dalvik / ART is nothing more than a basic constant value given in an app's manifest file [3]. When an app is installed by the Package Manager, the credentials of that app are applied to a database that includes permissions, which is in essence part of a larger database with packages containing a lot more useful information than just permissions such as public keys etc. Finally, what stands between VM level permissions and those of Linux and at the same time connecting them, is a file called *platform.xml* in */system/etc/permissions/* directory that actually maps a given named permission to a group or app ID.

Ultimately, it's worth mentioning the application signing and authentication system that Android makes use of. Without the need of complex frameworks and permissions, application signing helps developers recognize the author of the application and upgrade it. Any program running on the Android platform must be developer-signed. Either Google Play or the software installer on the Android computer would reject applications that try to run without signing them [34].

2.2. Android Malware Analysis

Nowadays almost every cyber threat contains malware and at the same time the number of such attacks is rising, while the attackers are becoming bolder. Every day, millions of bits of malware are seen, but usually not enough resources and time are out there to deal with it all. In most intrusion and vulnerability cases, malicious software, plays a critical role. Malware, including trojan horses, worms, rootkits, viruses and spyware, can be called any software that does anything that causes damage intentionally to a person, device, or network [34]. In order to learn how it operates, how to classify it, and how to beat or remove it, malware analysis is used, as the process or technique of evaluating the characteristics, origin and possible effect of a given sample of malware. Results from such analysis are used for multiple purposes, like incident response in order to have an overview of root cause, measure effect and excel in recovery and restoration, or even malware research to obtain an appreciation of the new methods, exploits and tools employed by malicious users.

With respect to the process of analysis, quite likely, often the analyst will have only just an executable, probably not human-readable. There will be also available a number of techniques and tricks to make sense of it, each disclosing a small amount of knowledge. In order to display the whole picture, you'll need to use a number of software and tools. On that matter, malware analysis has two basic approaches: static and dynamic. Static analysis is done by dissecting the various binary file resources without executing them and observing each variable. It is also possible to disassemble the binary file or reverse engineer it using a disassembler (tools like Ghidra and IDA, for reverse engineering and malware analysis). Sometimes the code can be transformed into assembly language that humans can read and understand, and so someone can make sense of the assembly instructions and get a clearer understanding of what the software is doing and how it was initially designed. Standard static analysis is easy and could be fast, but it is largely ineffective against advanced malware, and important behaviors and activities can be skipped [15]. Among the difficulties that are presented, are techniques that adversaries may use such as obfuscation, a process which makes it hard to understand textual and binary data. It allows malicious user to conceal crucial terms such as strings because they expose patterns of the actions of the malware.

Encryption and encoding methods are widely used by rivals to hide data in terms of obfuscation. Quite often, they go a little deeper and use special methods called packers which compress the executable into a packed file and self-extracts into memory in runtime, which make reverse engineering and research even more difficult [29].

On the other hand, dynamic analysis is done when the malware is already operating on a host machine by analyzing the malware's actions. In a sandbox setting, this type of inspection is often done to prevent the malware from actively infecting production processes; several such sandboxes are virtual structures that can be quickly rolled back to a clean state after the completion of the study. Current malware can present a wide range of evasive strategies designed to overcome complex analytics, including virtual environment testing or active debugging tools, slowing harmful payload execution. In this case, the provided solution to this, is to use a hybrid version of this by combining both of static and dynamic analysis by providing the best of these approaches.

Moving on from generic ideas on malware analysis to more specific environments like Android, we could say that mobile devices have an extremely different vulnerability environment, albeit comparable in some ways to desktops. Their mere versatility, unlike the other, exposes them to even more dangers, since they can be unintentionally lost or purposely stolen. This essentially negates the software security features that one could impose on a desktop system, by limiting access at the lock and key level, opening up a slew of threats that an attacker could attempt until a system has physical access. The attack profile has also changed since mobile devices are more likely to include personal user data and instead of taking complete control of the device centrally, it is always necessary to only access user data and smuggle it out to a remote server, potentially often using an internet connection.

Thus, in Android-powered devices the main threat vector is from within: that of an application [34]. A misbehaving or purposely malicious app may try to access the details of the user or even take over the functions of the device. Android must consider all apps as suspicious in order to avoid this. Applications are granted a minimum range of permissions by default, otherwise they are constrained. However, this range should not contain something that might be potentially vulnerable even if it is considered important. For this purpose, any extra permissions must be expressly asked in the application\s manifest for any approval outside of the minimum range. Each program is granted its own UID, which isolates it from others, and root access for software is never given, needless to say.

That, though, is not enough since Android must secure itself as well from the inside, since it is possible that a malicious application will attempt to target fragile components of the OS itself, given the small subset of permissions it has. Such vulnerabilities can be abused and more privileged operating system components can be fooled into executing an activity on behalf of the application, especially those running as root [34]. This is a major threat due to the large number of software running in the Android system, and even more code in the underlying Linux kernel.

Latest studies based on developments in mobile malware, suggest that the number of malicious Android applications is currently between 120,000 and 718,000 [8]. It is possible to categorize most Android malware into two forms, each using social engineering to manipulate users into running the malicious software. The bulk of malware for Android is known as fake installers. These applications claim to be an installer on their computers for legitimate applications and trick users into downloading them. The software will display a service agreement when executed and, after the customer has agreed, sends high rated text messages. Variants involve repackaged software that have the same features as the original version, mostly paid for, but have extra code in the background to secretly transfer SMS messages. SMS trojans are relatively simple to implement: it is only important to execute a single activity with a button that begins sending an SMS message when pressed [8]. Another Android malware form that has been found is known as spyware and has the potential to forward private data to a remote server. The malware may also obtain commands from the server to start specific operations in a more complicated manner, in which case it is part of a botnet. Broadcast receivers are of specific interest when they're used to remotely capture and forward incoming SMS messages to a remote server or wait until the system is launched to initiate a background operation. The advanced Eurograbber attack demonstrated in the summer of 2012 that this form of malware could be very profitable by stealing an estimated 36 million EUR from bank customers in Italy, Germany, Spain and the Netherlands [8].

Public access to known samples of Android malware is mainly supported by plenty of datasets found online. One of the most prominent datasets is the Drebin one, which includes 5,560 software from 179 separate families of malware. The samples were collected and made available by the MobileSandbox project during the period from August 2010 to October 2012. Another option is AndroZoo and DroidCollector, which are growing repositories of Android applications from various outlets, including the official store for Google Play apps.

2.2.1. Static Analysis

The Android static malware detection is the same as that of other OS. For Android malware, what varies is how APKs are packed and assembled in contrast to a normal binary. Android apps are installed as an APK that can be unpacked to contain the source code, a manifest, and other files similar to that, as separate. Without running any code, Android static analysis is based on analyzing an APK. Although it may theoretically expose all potential execution routes, there are many drawbacks. After all, all static approaches are susceptible to obfuscations such as encryption that eliminate or restrict access to the code [15]. Similarly, code injection or object changing at runtime are outside the static inspection framework since they are only observable during execution. *Application-Manifest.xml*, which explains permissions, names, libraries and application components like activities or services, along with *classes.dex*, which includes all of the application classes compiled into a compliant dex file format, are two important APK components for Android static analysis and identification.

There are several typical file data points that can be gathered automatically if you look at an Android app: filename, accessed times, size and type [15]. When searching for identical samples that may have special names or variations that may occur on other devices when managing an incident review, a suspect filename may be useful later. The more special a filename is, the more helpful it can be when correlating or checking. In corresponding to a threat, date and times associated with the file can also be helpful. An event, for instance, can include attacks that have occurred on or near a particular date. In certain cases, searching for threats of a certain kind, such as applications, matching times changed, or accessed, can help to discover other threats associated with static analysis installed in an attack. Also, file type is a type of document analysis where it is likely that the initial filename is misleading. Files are often not what they appear to be, like a file that claims to be a certain extension, but it's really something else. In this case, APK files are usually listed as ZIP (compressed) files.

First thing to do after getting done with the APK filename, extension, times and so on, is to unpack (or unzip) the APK, a trivial and simple process. Unpacked APKs, as already mentioned, include an *AndroidManifest.xml*, *classes.dex*, *resources.arsc* and META-INF directories. They will also include other libraries and assets directories, while the manifest XML formatted file includes the app's metadata and permissions, the key point to measuring the app 's capabilities.

Regarding the certificates and digests included in META-INF directories, all applications must be signed or they can't be installed [12]. Certificate details in some cases proved to be rich information source in the early days of Android attacks, since unauthorized developers were able to share codes easily without intervention inside approved marketplaces and did not change certificates. As a consequence, looking at certificate details helped (and can still sometimes help) researchers to compare and relate the same author's interests.

One of the most important parts of evaluating as part of this process, is the permissions. It is important to assert these demanded permissions inside the *AndroidManifest.xml* format. Since the manifest is simple to statically access, static analysis based on the permissions given, is used by many applications to determine the threats of the Android authorization scheme and individual apps. Most malware researchers accept that the Android authorization system's development tends to implement unsafe permissions and does not discourage malware from leveraging bugs and increasing them [34].

What's more, intents are abstract structures within Android that hold knowledge about an action to be done for an application component. The required behavior is taken by the device depending on the purpose and thus may be useful for study. In particular, it is possible to leak private data to a malicious app that requested the data through intents specified in its Android manifest file.

The mentioned hardware components are another aspect of the Android Manifest that has been used for static analysis. This can be useful when applications need to request all the hardware they need in order to work (e.g., microphone, GPS). Therefore, such combinations of demanded hardware can indicate maliciousness. For starters, there is no obvious need to demand GPS access for a contacts app.

Eventually, in the Android APK, the dex or classes.dex files can be found, clearly the key element for a static analysis. They are difficult to read for humans and are therefore first decompiled into a more readable format. There are several stages of formats, from low-level bytecode to assembly code to source code that can be read by humans. Most tools decompile dex files into assembly-like code (it is considerably important to mention that the most known form of assembly-like representation of bytecode is called Smali) while others prefer to acquire Dalvik byte code or even source code. In general, more forceful decompiling approaches have a higher failure rate or error rate, which can be changed by a form of post-processing. Also, features including classes, layout sequences, and application dependency diagrams can be extracted and evaluated from the decompiled code. On top of that, strings contained on extracted classes.dex files is equally valuable to find and record. That is why tools like the built-in executable *strings* is often used, to look into all the strings directly. While there are other alternatives, this is the most significant code part of an application and comprises the most valuable data of interest. When strings from classes.dex or other files are collected, they can be evaluated and examined with other tools in order to automatically extract any suspicious data that exists within the code.

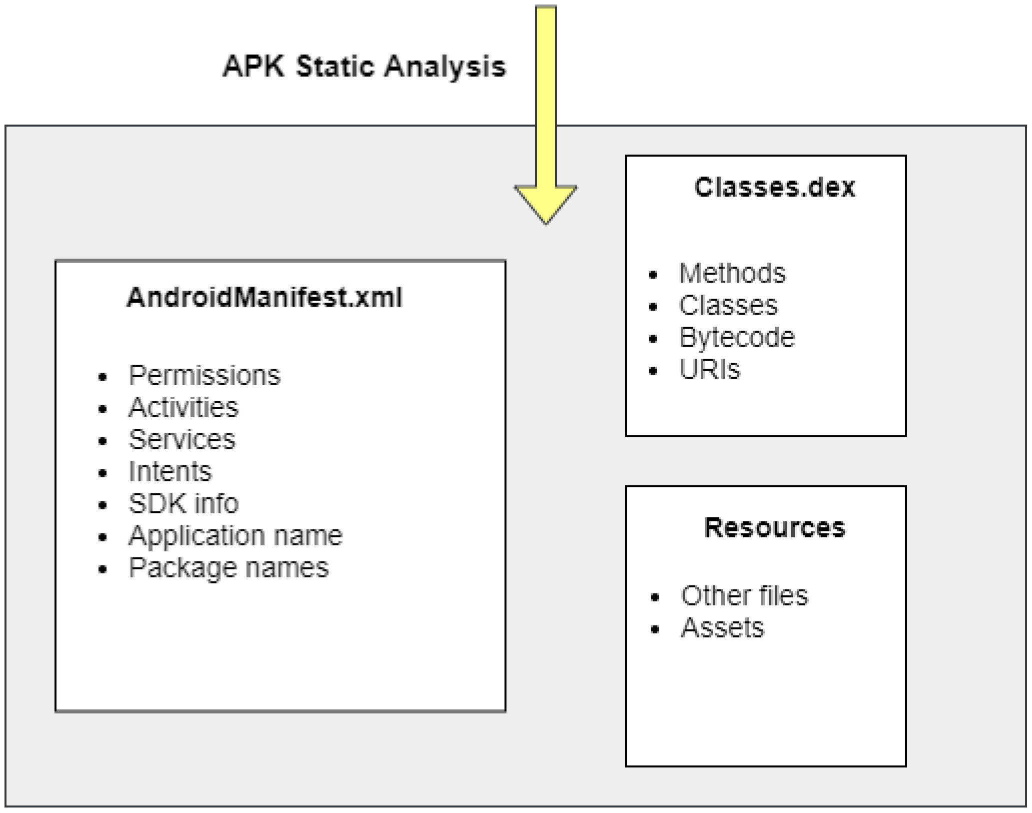


Figure 2.6. Android static analysis

With regard to static analysis tools used globally for Android malware analysis, we focus mostly on open source tools that utilize all the basic needs for such a process. For example, the *strings* executable command which is built into Linux is a very important part of any analysis.

It may include information associated with the construction of malware, accessibility and more. Once they are unpacked, the most important strings in an app are found in classes.dex, the actual source code. Other strings and data are also significant, but the app's source code obviously means the most. Also built into Linux, the *file* command is equally important. Any malware researcher can tell you that when it comes to code files, particularly an extension of a filename, there is nothing that can be trusted. This file executable helps to easily detect file types and show more info about them. Some information about the packages from Android is that they will appear as ZIP archives.

In the Java Development Kit, *Keytool* is installed and widely used to examine Android malware. *Keytool* prints an app with interesting detail, such as the country code, area, and more. In the early days of Android malware, this knowledge used to be indispensable to help correlate with unique rogue developers, but is typically faked in current malware.

For conventional Android malware researchers, *Dex2Jar* is a standard approach that transforms an app's DEX source code files to a JAR for a review of the converted Java code. Converted code analysis is not as accurate in some instances as studying source code inside the native Smali, but this kind of code analysis is more than satisfactory and very fast for most researchers in general. Moreover, in order to decompile and evaluate Java byte code, the Java Decompiler project is intended to give resources to do so. *JD-GUI* is a standalone interactive utility that presents .class file source Java codes. For quick access to methods and sectors, you can search the recovered source code with *JD-GUI*. JD-Core is also a library that, from one or more .class files, reconstructs the Java source code. JD-Core can be used to retrieve lost source code and to explore Java runtime library sources and is included by JD-GUI. *APKTool* is also a versatile approach that is widely explored by many. It is a widely recommended freeware application that provides APKs and XML decompilation. Production from the decompilation of APKTool results in simple to read permission / XML files and other useful data.

Besides manual decompilation, *AndroWarn* is a tool aimed primarily at detecting and alerting the user to potentially malicious activities generated by an Android application. The identification is achieved by static analysis with the *androguard* library of the application's Dalvik bytecode, described as Smali. Else way, *AndroTotal* is an excellent way to start a search of a problematic app as it offers a fast rundown of the static information and links to various useful third-party sites such as VirusTotal. Sites such as VirusTotal are traditionally mostly antivirus multiscanners.

2.2.2. Dynamic Analysis

Android dynamic (or also called behavioral) analysis is somewhat close to analyzing all other binary executables, not depending on the OS. It carries out its results by the use of normal dynamic testing, which means it achieves the behavior of the application through an application sandbox, virtual machine, or runtime emulation of the application. For complex malware identification, it performs real-time identification, tracks contact between suspicious IP addresses or uses packet sniffing. Current works that use dynamic analysis predominantly function during the runtime phase focused on application behavior analysis. Ordinarily, system calls, network utilization and battery use [30] are the key parameters that can be viewed and examined as the main behaviors of the application.

In more recent times, most of dynamic analysis happens through a list of automated tools. Most of them represent a modified version of the Dalvik Virtual Machine or ART from Android that provides an inclusive trace yield process. Each application automatic analysis reveals information about its actions, invoking parameters like return values, network correspondence records, internal method calls, executed Java code, user interactions, or even calls and messages [30]. They try to expose an app's malicious ways, simulate incidents and test their analytical effectiveness. Most of these tools also try to reflect applications' post-install activity on mobile devices and involve the application's behavior on the platform or on the network.

One of these automated tools is called DroidBox. It has been developed to provide interactive analysis of Android users. In the results created when the analysis is complete, it usually gives an output of hashes for the analyzed package, incoming / outgoing network data and leaks, file read and write operations, and SMS or phone calls.

It is solely based on the ART compiling system and during on-device compilation it tries to modify any application code. ARTist integrates especially well into the installation process of the Android app because it does not modify the APK, but instead removes the compiled native version, thereby retaining the package signature such that updates are always obtained by updated apps. Based on these two tools, we could add here that there is a huge list of behavioral analysis techniques and other automated ways for a researcher to do his job, at least helping them run and examine an application package.

Concerning testing methods now, the most crucial and important aspect of it, is a sandbox implementation or undoubtedly the development of an isolated environment, meaning it should be performed carefully and with appropriate preparation. The goal should be both to build a device capable of meeting all the needs needed and make it as practical as possible. Leaving deliberately traces of users such as browser cookies or documents can be considered an option, because you may be able to tell whether a malware is built to run, exploit or steal those data [30]. One of the most popular tools in this subject, is called CuckooDroid. It is an extension of the Cuckoo Sandbox platform for the automation of suspicious file analysis, while it adds the features of android program execution and analysis to the Cuckoo.

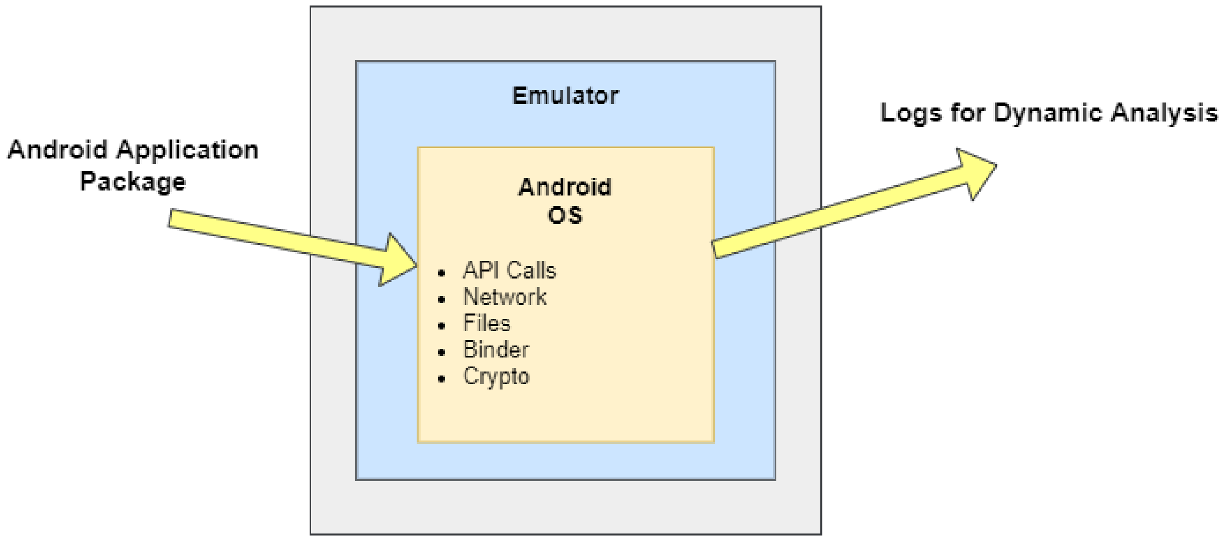
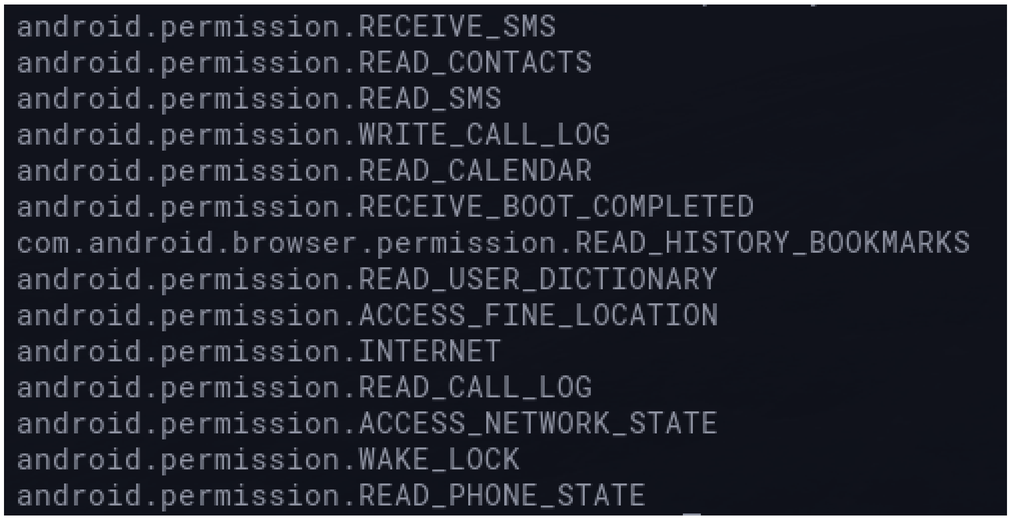


Figure 2.7. Android dynamic analysis

2.2.3. An example of Hybrid Analysis

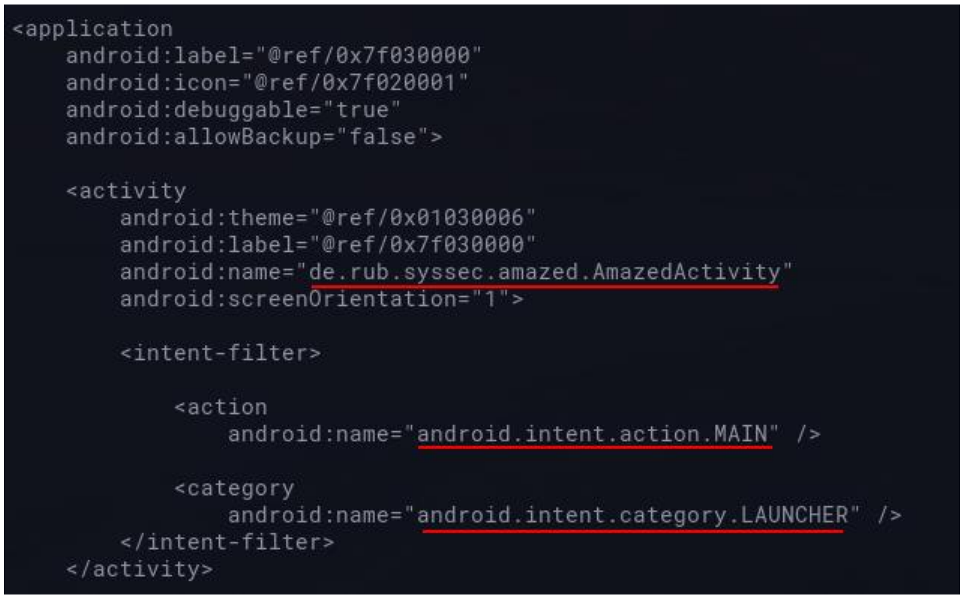
This illustration is a preview of malware called *syssecApp.apk* written for the Ruhr University-Bochum Reverse Engineering Summer School in 2013 and it offers a rundown of what Android malware is capable of doing. This apk contains the Amazed package / game found on Google play with extra malware code and is already given an example analysis in older walk-throughs found online, so it is quite good to be used as a sample since it uses no obfuscation. It is not attached to a remote controller, so it can never leave the attacked phone with the data it steals. However, some personal information would be evident in the logs and during this study.

At first, once downloaded, we get the MD5 and SHA-256 hashes for the sample malware, for future use. Then, we can begin with static analysis methods. The easiest thing to do, is to examine the *AndroidManifest.xml* file including the permissions required and all the entry points assigned to it, like activities and services. By examining the permissions, it can be observed that the application needs multiple different permissions like accessing the internet or bookmarks and reading or receiving SMS and call logs, which at first glance may seem suspicious. Having a look now at the entry points of the application, we can see that there are components that execute when the device receives an SMS or an alarm and when it boots up. The main activity representing the main on-screen app is de.rub.syssec.amazed.AmazedActivity.



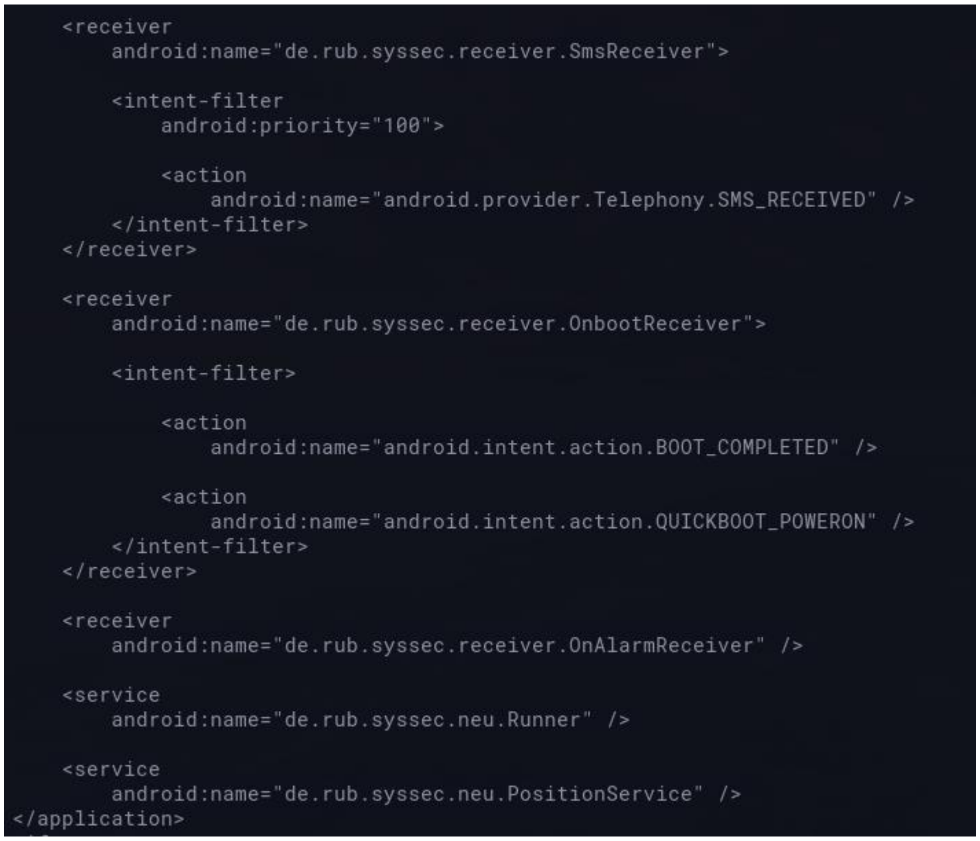
```
android.permission.RECEIVE_SMS
android.permission.READ_CONTACTS
android.permission.READ_SMS
android.permission.WRITE_CALL_LOG
android.permission.READ_CALENDAR
android.permission.RECEIVE_BOOT_COMPLETED
com.android.browser.permission.READ_HISTORY_BOOKMARKS
android.permission.READ_USER_DICTIONARY
android.permission.ACCESS_FINE_LOCATION
android.permission.INTERNET
android.permission.READ_CALL_LOG
android.permission.ACCESS_NETWORK_STATE
android.permission.WAKE_LOCK
android.permission.READ_PHONE_STATE
```

Figure 2.8. syssecApp permissions



```
<application
    android:label="@ref/0x7f030000"
    android:icon="@ref/0x7f020001"
    android:debuggable="true"
    android:allowBackup="false">

    <activity
        android:theme="@ref/0x01030006"
        android:label="@ref/0x7f030000"
        android:name="de.rub.syssec.amazed.AmazedActivity"
        android:screenOrientation="1">

        <intent-filter>

            <action
                android:name="android.intent.action.MAIN" />

            <category
                android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
```

Figure 2.9. syssecApp main activities

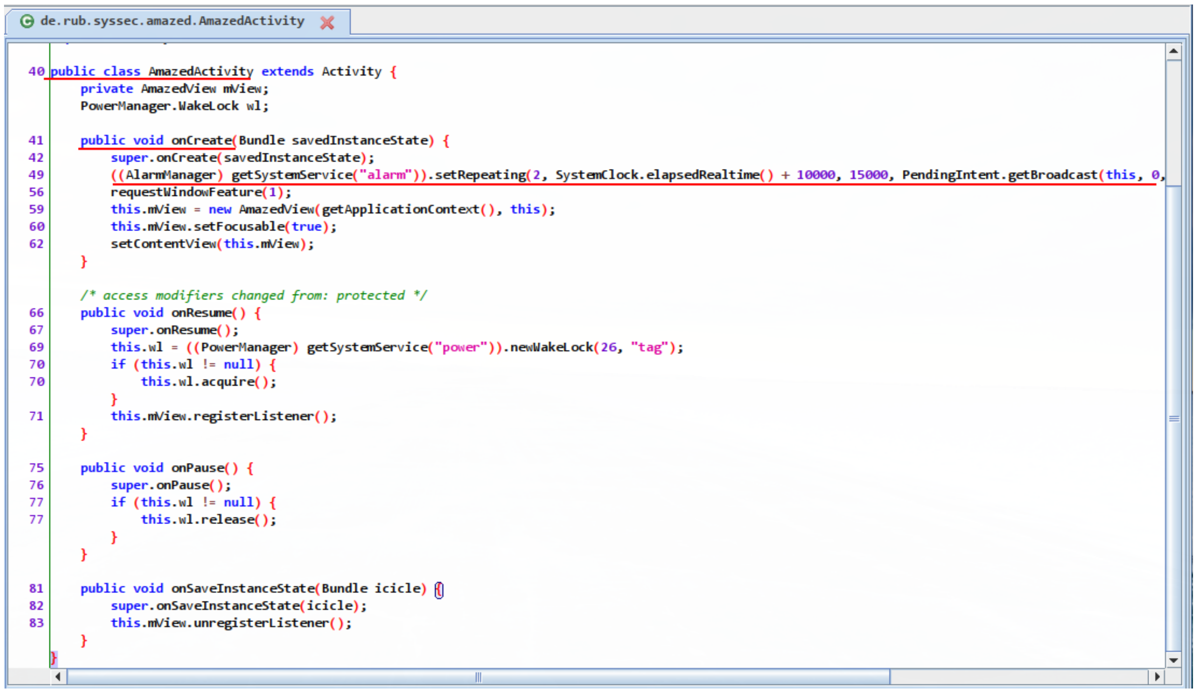


```
    <receiver
        android:name="de.rub.syssec.receiver.SmsReceiver">

        <intent-filter
            android:priority="100">

            <action
                android:name="android.provider.Telephony.SMS_RECEIVED" />
        </intent-filter>
    </receiver>

    <receiver
        android:name="de.rub.syssec.receiver.OnbootReceiver">

        <intent-filter>

            <action
                android:name="android.intent.action.BOOT_COMPLETED" />

            <action
                android:name="android.intent.action.QUICKBOOT_POWERON" />
        </intent-filter>
    </receiver>

    <receiver
        android:name="de.rub.syssec.receiver.OnAlarmReceiver" />

    <service
        android:name="de.rub.syssec.neu.Runner" />

    <service
        android:name="de.rub.syssec.neu.PositionService" />
</application>
```

Figure 2.10. syssecApp receivers/services

Our next step is to decompress the apk file and get the files inside. After that, we can run the *strings* command on the *classes.dex* file in order to grab any suspicious or useful strings contained in the file that can be also used later in dynamic analysis (now we can have more info on the structure, URLs found inside or even code functions). In addition to that, the core of our static analysis begins with examining the java code inside that file by using for example an inspecting tool with possible decompilation like *Jadx* (Dex to Java decompiler). Here, it has to be mentioned that using such a tool, may helps us find all the previous information about the apk with the help of a more sophisticated GUI instead of using commandline tools like *strings* or *apkanalyzer*. Having a deeper look at the onCreate method of the main Activity, it is noticed that it makes use of the system service *alarm* to set a recurring alarm every 15 seconds. More precisely, after getting the handle for this service, it creates an *AlarmManager* object along with an intent that is registered to run code as *OnAlarmReceiver*.

Then it objectifies a new *PendingInten*t entity that is intended to conduct a broadcast within the current app context that will execute the previous intent created. Finally, it sets a repeating alarm to perform the same broadcast every 15 secs after the first one.

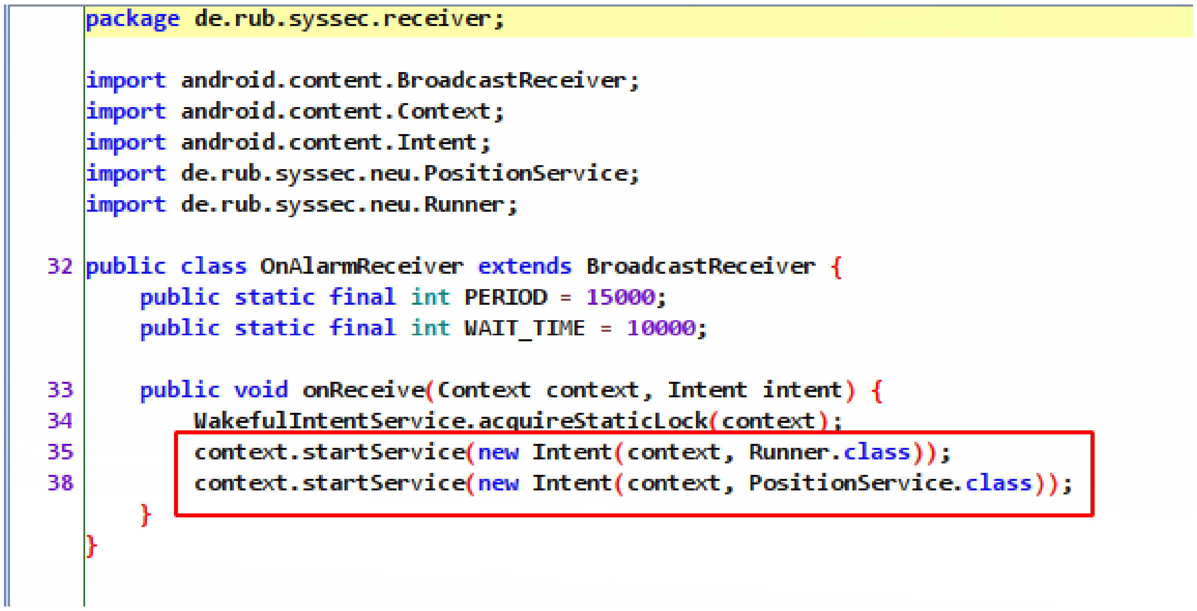


de.rub.syssec.amazed.AmazedActivity

```java
public class AmazedActivity extends Activity {
    private AmazedView mView;
    PowerManager.WakeLock wl;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ((AlarmManager) getSystemService("alarm")).setRepeating(2, SystemClock.elapsedRealtime() + 10000, 15000, PendingIntent.getBroadcast(this, 0,
        requestWindowFeature(1);
        this.mView = new AmazedView(getApplicationContext(), this);
        this.mView.setFocusable(true);
        setContentView(this.mView);
    }

    /* access modifiers changed from: protected */
    public void onResume() {
        super.onResume();
        this.wl = ((PowerManager) getSystemService("power")).newWakeLock(26, "tag");
        if (this.wl != null) {
            this.wl.acquire();
        }
        this.mView.registerListener();
    }

    public void onPause() {
        super.onPause();
        if (this.wl != null) {
            this.wl.release();
        }
    }

    public void onSaveInstanceState(Bundle icicle) {
        super.onSaveInstanceState(icicle);
        this.mView.unregisterListener();
    }
}
```

Figure 2.11. syssecApp decompilation

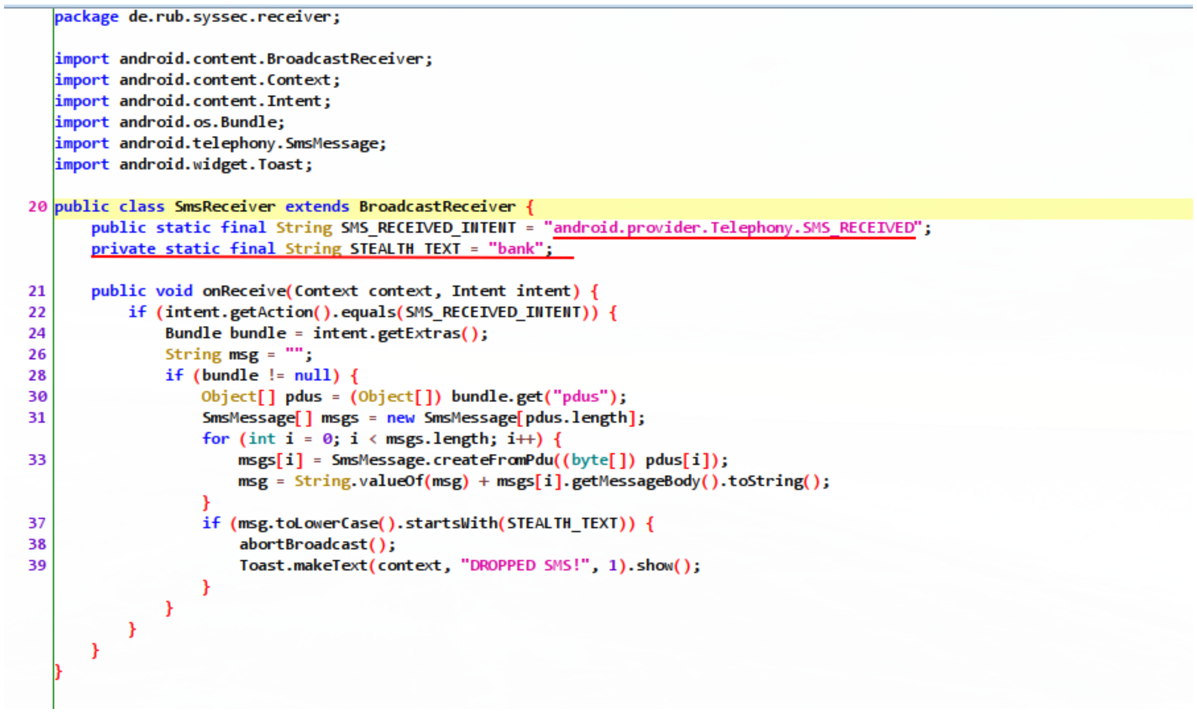
Searching more inside inside main activity's components, we take a look at *onAlarmReceiver* broadcast receiver. There, it is seen that this class is responsible for starting two new services (*Runner* and *PositionService*) when receives a broadcast message. Inside Runner class, there are a lot of suspicious methods created like *steal*, *sendData* or *read*. After reading some of *steal* method's code, we understand that it is about the extra malware code that reads from Calendar, Call logs, Browser, etc. and sends it in a remote address (which is 127.0.0.1 in this case). Based on the API calls in the method list, the information gathered by the app is as follows: · IMSI · SIM serial number · Computer ID · Contacts · Call logs · Calendar info · Web cookies and bookmarks · SMS and finally GPS info such as Location.



```
package de.rub.syssec.receiver;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import de.rub.syssec.neu.PositionService;
import de.rub.syssec.neu.Runner;

public class OnAlarmReceiver extends BroadcastReceiver {
    public static final int PERIOD = 15000;
    public static final int WAIT_TIME = 10000;

    public void onReceive(Context context, Intent intent) {
        WakefulIntentService.acquireStaticLock(context);
        context.startService(new Intent(context, Runner.class));
        context.startService(new Intent(context, PositionService.class));
    }
}
```

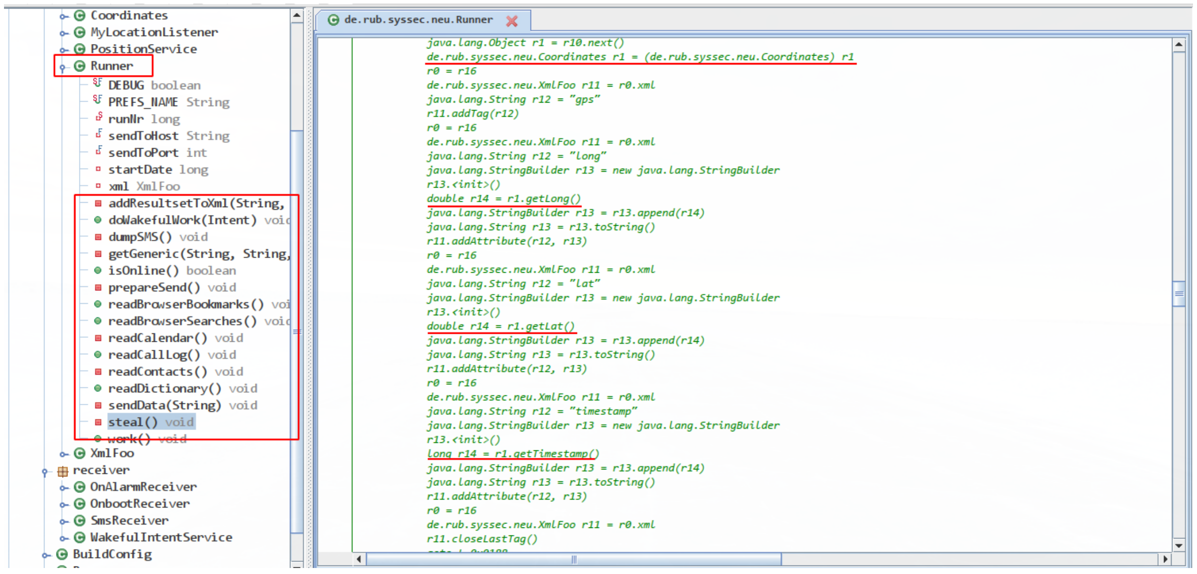
Figure 2.12. syssecApp onAlarmReceiver

Apart from *onAlarmReceiver* that are also two other Receiver classes called *onBootReceiver* and *SmsReceiver*. The first one contains multiple event handlers such as *onBoot* handler which sets the alarm since booting, while the latter is called each time an SMS is obtained by the app and it verifies if the message begins with the word bank.



```java
package de.rub.syssec.receiver;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.telephony.SmsMessage;
import android.widget.Toast;

public class SmsReceiver extends BroadcastReceiver {
    public static final String SMS_RECEIVED_INTENT = "android.provider.Telephony.SMS_RECEIVED";
    private static final String STEALTH_TEXT = "bank";

    public void onReceive(Context context, Intent intent) {
        if (intent.getAction().equals(SMS_RECEIVED_INTENT)) {
            Bundle bundle = intent.getExtras();
            String msg = "";
            if (bundle != null) {
                Object[] pdus = (Object[]) bundle.get("pdus");
                SmsMessage[] msgs = new SmsMessage[pdus.length];
                for (int i = 0; i < msgs.length; i++) {
                    msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
                    msg = String.valueOf(msg) + msgs[i].getMessageBody().toString();
                }
                if (msg.toLowerCase().startsWith(STEALTH_TEXT)) {
                    abortBroadcast();
                    Toast.makeText(context, "DROPPED SMS!", 1).show();
                }
            }
        }
    }
}
```

Figure 2.13. syssecApp smsReceiver

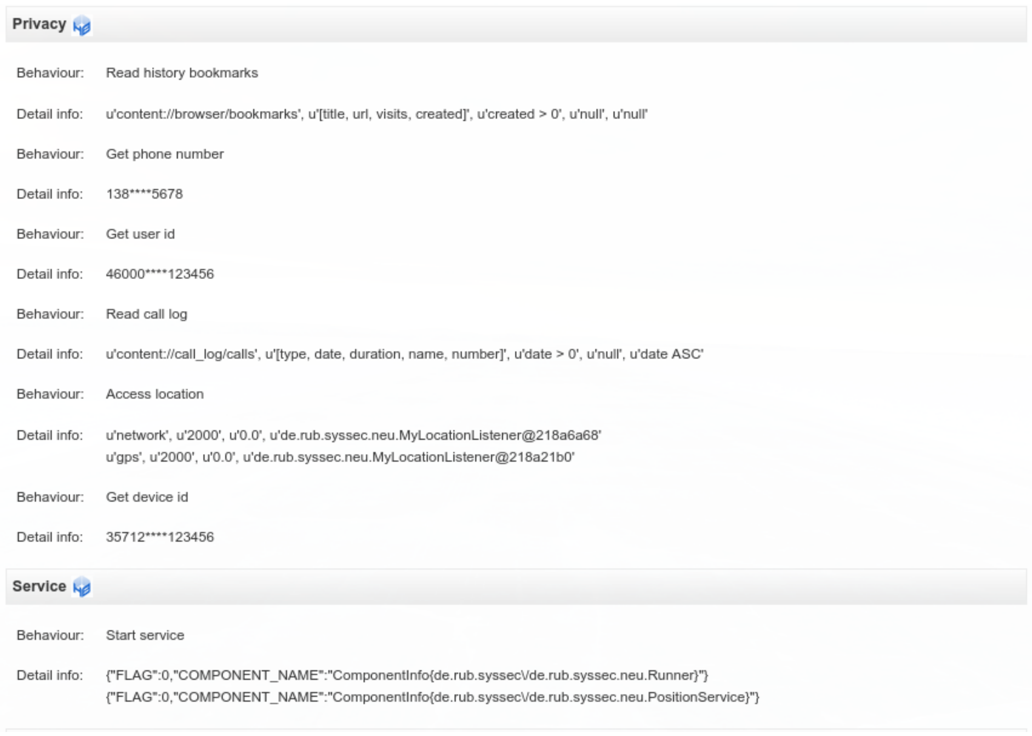


```
Coordinates
MyLocationListener
PositionService
Runner
    DEBUG boolean
    PREFS_NAME String
    runNr long
    sendToHost String
    sendToPort int
    startDate long
    xml XmlFoo
    addResultsetToXml(String,
    doWakefulWork(Intent) voi
    dumpSMS() void
    getGeneric(String, String,
    isOnline() boolean
    prepareSend() void
    readBrowserBookmarks() voi
    readBrowserSearches() voi
    readCalendar() void
    readCallLog() void
    readContacts() void
    readDictionary() void
    sendData(String) void
    steal() void
    work() void
XmlFoo
receiver
    OnAlarmReceiver
    OnbootReceiver
    SmsReceiver
    WakefulIntentService
BuildConfig
```

```
de.rub.syssec.neu.Runner

java.lang.Object r1 = r10.next()
de.rub.syssec.neu.Coordinates r1 = (de.rub.syssec.neu.Coordinates) r1
r0 = r16
de.rub.syssec.neu.XmlFoo r11 = r0.xml
java.lang.String r12 = "gps"
r11.addTag(r12)
r0 = r16
de.rub.syssec.neu.XmlFoo r11 = r0.xml
java.lang.String r12 = "long"
java.lang.StringBuilder r13 = new java.lang.StringBuilder
r13.<init>()
double r14 = r1.getLong()
java.lang.StringBuilder r13 = r13.append(r14)
java.lang.String r13 = r13.toString()
r11.addAttribute(r12, r13)
r0 = r16
de.rub.syssec.neu.XmlFoo r11 = r0.xml
java.lang.String r12 = "lat"
java.lang.StringBuilder r13 = new java.lang.StringBuilder
r13.<init>()
double r14 = r1.getLat()
java.lang.StringBuilder r13 = r13.append(r14)
java.lang.String r13 = r13.toString()
r11.addAttribute(r12, r13)
r0 = r16
de.rub.syssec.neu.XmlFoo r11 = r0.xml
java.lang.String r12 = "timestamp"
java.lang.StringBuilder r13 = new java.lang.StringBuilder
r13.<init>()
long r14 = r1.getTimestamp()
java.lang.StringBuilder r13 = r13.append(r14)
java.lang.String r13 = r13.toString()
r11.addAttribute(r12, r13)
r0 = r16
de.rub.syssec.neu.XmlFoo r11 = r0.xml
r11.closeLastTag()
```

Figure 2.14. syssecApp suspicious functions

Regarding dynamic analysis we could use a tools like DroidBox or CuckooDroid that use an emulator in order to display in an automated way information about incoming/outgoing network traffic, started services, file reads/writes, sent SMS or calls and any information leaks or even run the malware in our manner. In this case, we analyze the apk through an online website that offers free analysis services (both static and dynamic, in such matter we only need the dynamic analysis) named VirusTotal, just for the sake of this example. After submitting the file, the website gives back a report with details such as network behavior, dangerous functions, privacy behaviors, services started and other malicious intents. Furthermore, the detection service that VirusTotal offers showed that 26 of 64 antivirus engines find syssecApp as malicious, something that is visible from the following figures.



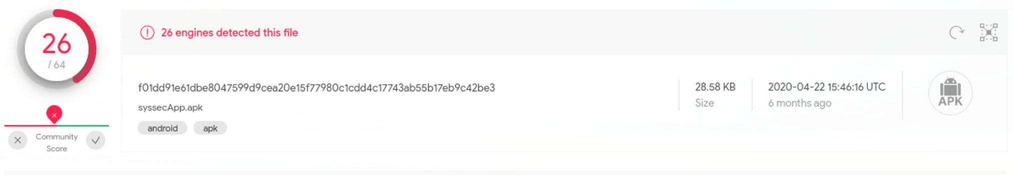
Privacy

Behaviour:	Read history bookmarks
Detail info:	u'content://browser/bookmarks', u'[title, url, visits, created]', u'created > 0', u'null', u'null'
Behaviour:	Get phone number
Detail info:	138****5678
Behaviour:	Get user id
Detail info:	46000****123456
Behaviour:	Read call log
Detail info:	u'content://call_log/calls', u'[type, date, duration, name, number]', u'date > 0', u'null', u'date ASC'
Behaviour:	Access location
Detail info:	u'network', u'2000', u'0.0', u'de.rub.syssec.neu.MyLocationListener@218a6a68' u'gps', u'2000', u'0.0', u'de.rub.syssec.neu.MyLocationListener@218a21b0'
Behaviour:	Get device id
Detail info:	35712****123456

Service

Behaviour:	Start service

Detail info:	{"FLAG":0,"COMPONENT_NAME":"ComponentInfo{de.rub.syssec\/de.rub.syssec.neu.Runner}"}
	{"FLAG":0,"COMPONENT_NAME":"ComponentInfo{de.rub.syssec\/de.rub.syssec.neu.PositionService}"}



26 / 64

Community Score

26 engines detected this file

f01dd91e61dbe8047599d9cea20e15f77980c1cdd4c17743ab55b17eb9c42be3
syssecApp.apk
android apk

28.58 KB
Size

2020-04-22 15:46:16 UTC
6 months ago

APK

Figure 2.15. syssecApp virusTotal analysis

2.3. Machine Learning

These days, apart from a heavy use of manual malware analysis in Android, an approach that uses machine learning is deliberately showing up. Machine learning is the analysis of computational algorithms that over experience, improve automatically [9]. It is used as an artificial intelligence sub-set. In order to test hypotheses or predictions without even being directly programmed to do so, machine learning algorithms create a model based on sample data, known as training data. In a broad range of applications, machine learning algorithms are used where the development of traditional algorithms to perform the necessary tasks is difficult or unworkable. Computational statistics, which concentrate on making calculations using machines, are closely linked to a branch of machine learning. The analysis of mathematical optimization provides the field of machine learning with techniques, theory and implementation domains, while data mining is a similar area of research, based through unsupervised learning on exploratory data processing [19].

Centered on the topic above, machine learning can provide an effective alternative to the traditional engineering flow where implementation expense and time are the key concerns, or when the problem seems to be too difficult to be examined in its full generality. On the contrary, this method has usually the main pitfalls of delivering suboptimal results, or hindering the solution's usability, and of applying it only to a small number of challenges [27]. The following parameters was proposed in order to classify activities for which machine learning techniques can be useful. For example, if the task includes a mechanism comprising input-output pairs that maps well-defined inputs to well-defined outputs or large data sets, whereas the process gives immediate input with clearly defined objectives. If the process does not include lengthy logical or rational chains that rely on various context information or do not include thorough reasons about how the decision was made or even if he task has error tolerance and no requirement for proven right and optimal solutions.

Based on their engagement with the experience or world or whatever we like to call the input data, there are various ways an algorithm can model a problem. In machine learning and artificial intelligence literature, it is common to first examine the types of learning that an algorithm should follow. This categorization or method of arranging algorithms for machine learning is beneficial because it allows you to reflect about the functions of the input data and the process of model preparation.

Starting with the category of supervised learning, input material is called training data and has a recognised label at a time, such as spam or a stock price [9]. A model is prepared by a training phase in which predictions need to be made and when those predictions are incorrect, it is corrected. The testing process continues until a desired degree of precision is reached on the training samples by the algorithm. Classification and regression are exemplary topics. Classification is when data is used to simulate a categorical attribute and it is often called supervised learning. This is the situation when a tag or identifier is given to an image, which is called binary labeling because there are just two categories to classify. If more than two groups exist, the emerged challenges are referred to as multi-class grouping. On the other hand, when the need for estimating continuous and constant values becomes apparent, these issues become a matter of regression.

On the opposite, using unsupervised learning, input data is not classified and doesn't have a known outcome. By comprehending constructs existing in the input data, a model is formed [19]. This could be for general standards to be extracted. It may be to minimize duplication systematically by a statistical method, or it may be to group data through similarity. Clustering and dimension reduction are examples of this kind of challenges. Clustering means grouping a number of data samples such that, according to certain requirements, examples in one group are more comparable than those in other categories. This is also used for the entire dataset to be segmented into many categories. In each category, research can be done to allow users to identify inherent patterns. In addition to that, reducing the dimension implies lowering the number of variables under consideration. The raw data has very high dimensional characteristics in certain applications and certain characteristics are redundant or unrelated to the task. Dimensionality reduction helps to uncover the real, implicit connection.

There exists also semi-supervised learning that means input data is a combination of examples that have labels or not. There is a required prediction problem, but the model must study the systems and make predictions to arrange the data. Classification and regression are noteworthy subjects. With supervised learning, the difficulty is that it can be costly and time consuming to mark data. You may use unlabeled examples to boost supervised learning if labels are constrained. Since the machine in this case is not completely supervised, we can say that the machine is semi-supervised. For semi-supervised learning, to increase the accuracy of learning, we use unlabeled samples with a limited amount of labeled data.

Finally, using reinforcement learning, a computer algorithm deals with a complex world in which a specific objective, such as driving a car or playing games against an adversary, must be achieved. The software is given input that is equivalent to awards, which it seeks to optimize, as it navigates its problem space. So, reinforcement learning evaluates and improves an agent's actions based on the environment's feedback. Instead of being told which actions to take, algorithms try various simulations to discover which actions produce the greater benefit. Reinforcement learning is differentiated from other methods by trial-and-error and delayed compensation.

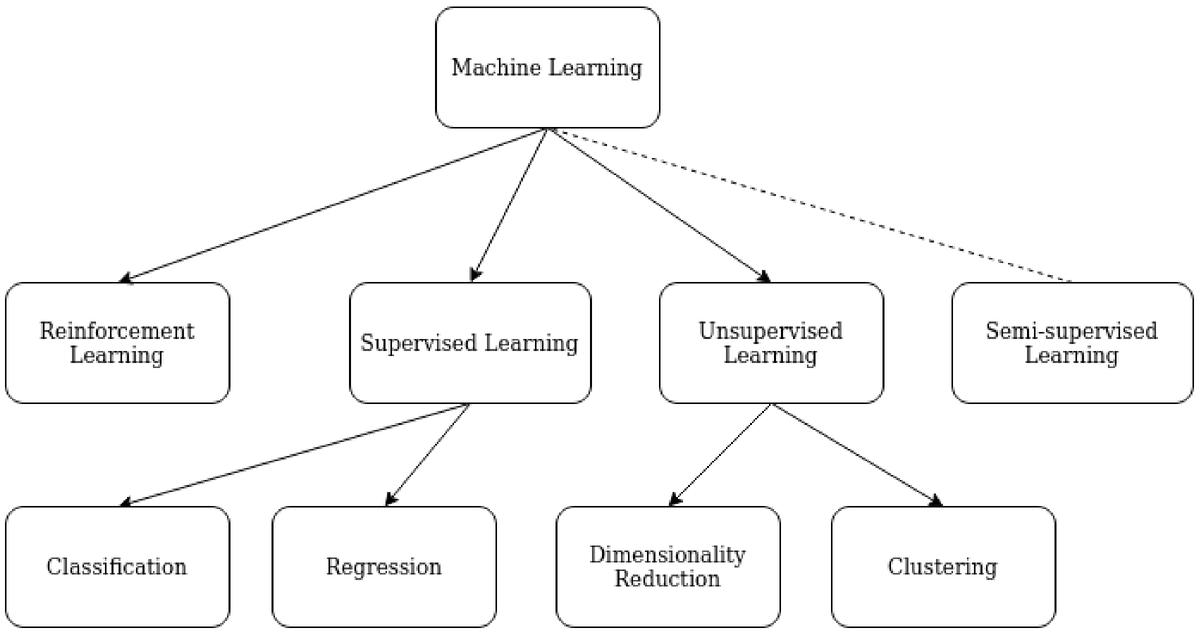


Figure 2.16. Types of ML algorithms

Often we take things like precision, learning time and simplicity of use into consideration, when selecting an algorithm. Many users put consistency first while beginners prefer to rely on algorithms that they know best [5]. The first thing to remember when faced with a dataset is how to produce outcomes, no matter what those results might look like. Beginners prefer to select easy-to-implement algorithms that can easily obtain results.

This works well as long as the method is just the first step. Performing machine learning requires developing a model, which is conditioned on certain training data and then can process additional data to create predictions. For machine learning applications, various kinds of models have been used and studied. Models also are used to categorize algorithms that are usually grouped by similarity, in terms of their role in how they run [2].

Input data to train a model to estimate outcomes are given in supervised machine learning. A few important concepts are provided in order to be able to describe the various models and how machine learning operates in a more comprehensive manner. First of all, the already existing classified data set needs to be divided into two sets in order to train the model and better assess its performance: a test set and a testing set. Both two sets must be distinct of one another, extracted from the initial data set arbitrarily. On top of that, there is cross-validation, which is a method of testing and performance assessment in order to eliminate the inherent variance that could arise due to a forced selection of samples being the test and training sets. The data set is divided into n equivalent, distinct, and random sub-sets. As a test set, one of the subgroups is selected and all other subsets are combined into one training set. Train the classifier, create estimates and analyze efficiency. Repeat the procedure by selecting the next subset as a training set each time and combining the other subsets into a training set every time.

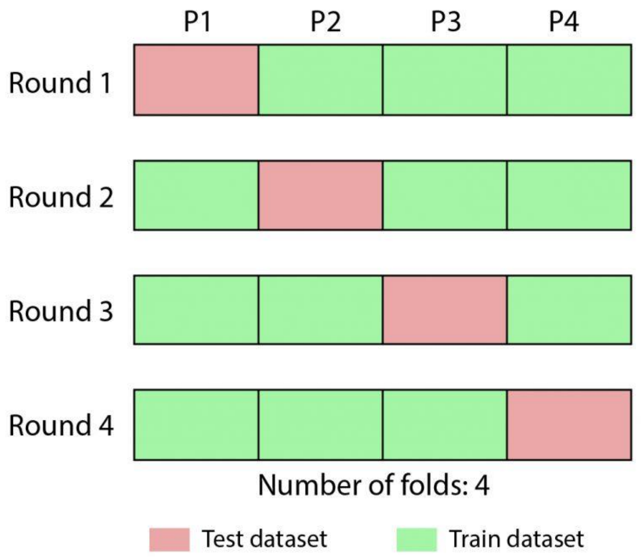


Figure 2.17. Cross-validation

2.3.1. Basic Algorithms

Regression is associated with modeling the relationships among variables that is adaptively optimized in the predictions produced by the algorithm using a measure of error. Regression approaches are a mathematical mainstay that have been co-opted into machine statistical learning [5]. This can be misleading, since we can use regression to refer to the problem class and the algorithm class. In order to approximate the relationship between input variables and their related characteristics, regression analysis involves a wide range of statistical approaches. Linear regression, where a continuous line is drawn to better match the data set as per a mathematical standard such as ordinary least squares, is the most common form. By fitting the best line, we create a relationship between different variables. This best suitable line is referred to as a line of regression and is described by a linear equation Y= a*X+b. Here Y is a dependent variable, A a slope, X is the independent variable and B is called intercept. These a and b coefficients are calculated on the basis of minimizing the sum of the square distance gap between both the pieces of data as well as the line of regression. The above is also generalized by statistical techniques of regularization to minimize overfitting.

Several other methods involve polynomial regression, logistic regression, sometimes used in statistical classification, or also kernel regression when working with non-linear issues, which adds non-linearity by using the kernel trick to indirectly map input data to higher-dimensional space. Logistic regression, given a particular number of independent variables, it is used to determine discrete or binary values like 0 and 1, true or false. It calculates, in simple terms, the likelihood of an occurrence happening by fitting data to a logistic function. It is thus known as logistic regression and it used for both classification and regression. Since the likelihood is estimated, the production values are between 0 and 1. The benefits of logistic regression are that it is very efficient, does not require large computing assets, does not need to scale data features it is quick and easy to apply. In contrast, non-linear situations are not handled correctly and until all independent variables are explicitly defined, it does not function effectively [9].

Support vector machines also called SVM, is a group of linked supervised learning methods intended for classification and regression. Provided a number of training cases, each labeled as belonging to one of two categories, an SVM training algorithm constructs a model that can predict if a new iteration fits into one group or the other.

An SVM trained model is a non-probabilistic, discrete, linear classifier, but in a probabilistic classification environment, there are methods that also use SVM [9]. In order to select the optimal fit to make a decision, such approaches usually build up a profile of example data and compare fresh information into the database using a correlation matrix. Example-based approaches are often called winner-take-all methods and memory-based learning, for this purpose. Emphasis is focused on the portrayal of the instances stored and metrics of similarity used between instances. In SVM models, kernel techniques are often used to convert non-linearly separable functions to higher dimension functions, often using the word "hyperplane". A training algorithm for SVM considers the classifier represented by the hyperplane's feature vector w and bias b. This boundary actually divides multiple classes by as large a margin as possible. The issue can be turned into a restricted question of optimization [19]. SVM essentially draws distinctions between instances of data plotted in multidimensional space features, being used to identify instances of data belonging to various classes. SVM's benefits are that it is versatile and that the outcomes of the estimation are very precise. On the other hand, it is only appropriate to binary classification while it is very complicated and slow with large datasets.

A model of assumptions based on specific properties of attributes in the data is built by decision tree methods, used both for classification and regression. Calculations fork in trees before a decision for a class label is made. In machine learning, decision trees are always quick, efficient and a major favorite, being used quite often in statistics and data mining. This type of learning, substantially, uses a decision tree to get from the findings on an element defined in its branches to assertions about the target value of the item, represented in the leaves as a prediction [5]. Tree models that can take a distinct set of values from the target variable are also called classification trees. Decision trees where continuous values such as real numbers are being taken by the target variable are considered regression trees. There are several types of decision trees, but they all have the same functionality, which is split the area of features of the function. Decision trees are quick to grasp and to execute, but when the branches are overloaded and they trees go far into them, however, they seem to hyper fit results.

Specifically now, random forest is a composite classification algorithm which institutes several classifiers of the decision tree and utilizes them. From the random collection of a sub-group within a data set, a set of decision trees is generated. Most votes are added to combine the output of the various trees when the random forest is produced with the mixture of decision trees.

The advantages of Random Forest would be that it functions well enough on broad and extensive data sets, includes procedures for mitigating errors in an unstable class population data set, has a reliable approach for calculating incomplete information and preserves accuracy in the absence of a large proportion of data. The drawbacks are that when it incorporates many decision trees, it is very hard to explain, it is far more costly than a simple decision tree model [19].

Clustering determines the problem class and the process type. Usually, clustering techniques are coordinated by modeling strategies such as centroid and other hierarchical approaches [5]. In order to better arrange the data into classes of maximum commonality, both approaches are concerned with utilizing the underlying constructs in the data. K-Means is a clustering method that uses grouping in machine learning based on segmentation. N data points are allocated to one of the K clusters of the K-means classifier, where K is a user-defined variable with the necessary clusters. In general, for each cluster known as centroid, K-means captures k number of points. A cluster with the nearest centroids, i.e. k clusters, is formed from each piece of data. It then finds each cluster's centroid dependent on current member nodes and repeats the previous steps since there are new centroids. For each data point, finds the nearest distance from new centroids and gets connected to new k-clusters. It finally repeats this step until there is enough stability, i.e. no shift in centroids. The advantage of the K-means classifier is that it is quite scalable and seems to be very strong in efficiency and results in several situations and complications. The drawbacks of clustering K-means are that it requires a random chance and in some situations may not be an ideal set of a cluster while at the same time certain skill is needed for the user to guess the initial number of natural clusters to have successful results.

Bayesian approaches are those which extend Bayes's theorem specifically to situations like classification or regression and one of them being quite popular is Naive Bayes. A Naive Bayes classifier claims, in basic terms, that the inclusion of a certain element in a category is irrelevant to the existence of any other feature. Even though these characteristics rely on one another and the presence of the other characteristics, all of these properties will be considered by a naive Bayes classifier to individually add to the possibility to classify a target. Naive Bayesian classifiers are direct linear classifiers and are recognized for their simple and precise outcomes [2]. The benefits of this classifier are that it is quick and efficient in computation, works well with larger and smaller quantities of data for training and it is relatively easy to perform well, in circumstances where it is noisy and lacking data.

On the other hand, if the set of data represents a significant amount of numerical characteristics, the presumption of equal importance and autonomy does not hold true as the precision and consistency of the output becomes limited [2].

2.3.2. Ensemble Learning

Ensemble approaches in statistics and machine learning employ multiple diverse learning algorithms to achieve greater predictive accuracy than either of the component learning algorithms individually. A machine learning ensemble is made up of only a limited number of alternative models, but it usually allows for a much more versatile design to arise among them. Trying to assess an ensemble's prediction usually necessitates further computation than estimating a single model's prediction. Ensemble learning can be viewed as a way to adjust for bad learning algorithms by doing a large amount of extra calculation, which means that the other option, on the other hand, is to do a lot of learning on a single non-ensemble scheme. An ensemble system could be more effective at enhancing average accuracy with the same improvement in computational and storage capacity by distributing the increase across multiple techniques than a single approach may have been [21]. Strong algorithms, such as trees, are often used in ensemble methods, but slower algorithms may also benefit from them.

Furthermore, where there is a lot of variety among the models, ensembles seem to produce better performance. As a result, several ensemble approaches aim to encourage diversity within the models they integrate. More random algorithms may yield a better ensemble than very intentional algorithms, which may seem counterintuitive. Although the amount of component classifiers in an ensemble has a significant effect on prediction accuracy, there are few studies that address this issue. The importance of evaluating ensemble size a priori, as well as the frequency and speed of large data sources, is much more important for online ensemble classifiers. The correct number of components was often determined using statistical tests [21].

There are multiple kinds of ensemble, but two are the one that distinguish the most, called averaging and boosting. The guiding force behind averaging methods is to construct many estimators independently and then average their forecasts. Since its uncertainty is minimized, the combined estimator is normally better than either of the single basis estimators. In boosting methods, on the other hand, base estimators are constructed sequentially and the cumulative estimator's bias is reduced.

Bagging methods (kind of averaging) are a type of ensemble algorithm that creates multiple cases of a black-box assessor on randomized subsets of the initial training set, then aggregates their individual predictions to shape a final prediction [17]. These techniques are used to reduce a base estimator's variance by integrating random sampling into the building process. In certain ways, bagging approaches are an easy way to develop with respect to a particular model without having to change the underlying base algorithm. Bagging methods, in comparison to boosting methods, which work better with poor models, work best with solid and complicated models so they offer a means to minimize overfitting.

On the other hand, boosting is a group of machine learning algorithms that transform weak classifier to intelligent ones. It is an ensemble meta-algorithm for specifically minimizing bias and even variance in supervised learning. Although boosting is not computationally limited, most boosting algorithms involve learning weak learners iteratively and then combining them with a final strong classifier. They are weighted in a way that is related to the precision of the poor learners when they are applied. The data weights are rebalanced after a slow learner is inserted, a process known as reweighting [7]. Input evidence that has been incorrectly categorized gains weight, while examples that have been correctly classified lose weight. As a result, potential weak learners will place a greater emphasis on instances that past weak learners incorrectly classified.

As an example of ensemble learning, two of the most popular algorithms used nowadays is Random Forest and Gradient Tree Boosting. Random forests, as mentioned before, are an ensemble learning tool for classification or regression that works by building a large number of decision trees during training and then extracting the class that is the average class of the individual trees. That said, gradient boosting is a machine learning method for regression and classification problems that generates a prediction model from an ensemble of poor prediction models, usually decision trees. The subsequent signal is considered gradient boosted trees when a decision tree is the weak learner, and it often outclasses random forest.

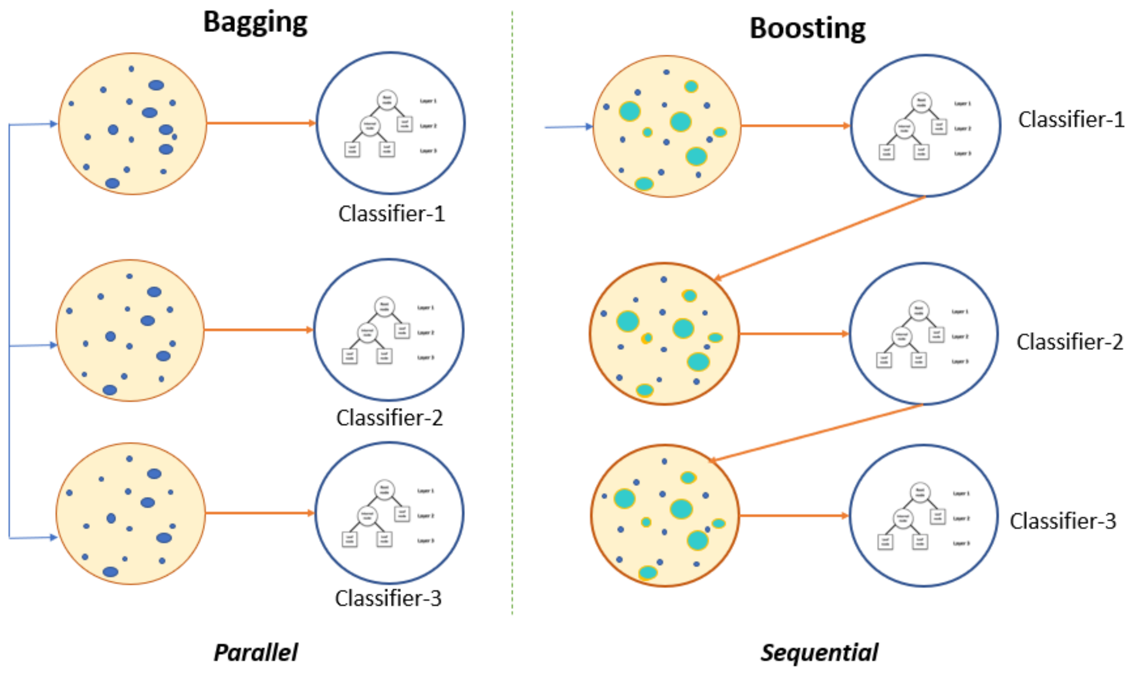


Figure 2.18. Boosting and Bagging

2.3.3. Neural Networks

Artificial neural networks are models which are based on biological neural networks' system and components. They are a method of template matching, frequently used during issues with regression and classification, but they are really a huge subfield consisting of a large number of algorithms and different versions for all kinds of classification methods [9]. In early days, neural networks succeeded because of their parallel and distributed computing capacity. But the inefficiency of the back-propagation training algorithm, which is commonly used to refine the variables of neural networks, has hindered research in this area. In machine learning, support vector machines as well as other simplified models that can be efficiently trained by addressing optimization problems have steadily replaced neural networks. In recent times, modern and enhanced training methods have fostered the growth of enthusiasm in neural networks, such as unsupervised pre-training and layer-wise training. This restored implementation has also been spurred by increasingly strong computing capacities, such as GPUs. The development of structures of thousands of layers, has been the result of resurgent studies in neural networks.

In other words, shallow neural networks have developed into neural networks of deep learning. Deep neural networks for supervised learning have been very effective, while extended to unsupervised learning functions, such as feature extraction, often removes functionality with far less human interference from raw images or expression [5]. They can model non-linear interactions that are complicated and create conceptual models in which the object is expressed as primitives' structured representation. The extra layers make it easier to compose features from lower layers, theoretically modeling complex information with fewer units than a shallow network that works in a similar way. Usually, these networks are a type in which information goes without crossing back from the input layer to the output layer. They initially generate a map of simulated neurons and apply relations between them to arbitrary numerical values, or weights. Weights and inputs are multiplied and an output of 0 to 1 is returned [9]. An algorithm would change the weights if the network did not correctly identify a given pattern. Other popular deep learning algorithms include convolutional deep neural networks (CNNs) that are used in image or automated speech recognition and recurrent neural networks (RNNs) in which information can move in either way, for applications such as language processing.

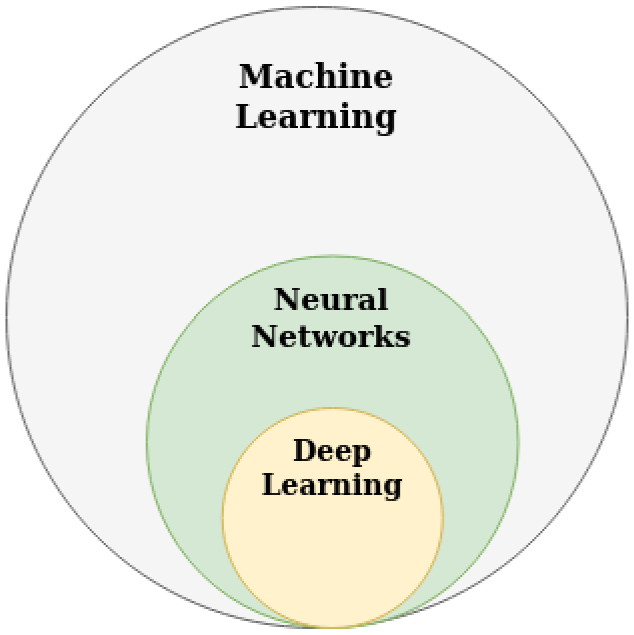


Figure 2.19. Neural Networks

2.4. Related Work

With the explosive rise of Android malware, many methods have been developed to identify the OS's inherent vulnerabilities. Static and dynamic analysis are the two main types of approaches for ensuring and reviewing Android apps. Here follows a section that describes the fundamental work and research projects, containing topics about Android malware analysis and machine learning, that aided our thesis.

Static analysis was the first Android malware identification method, in which specific pieces of code were examined without the program being run on an external computer or an Android emulator. It is an automatic procedure that provides an Android app's source code or binary file, analyses it without running it, and generates analysis results by testing the code structure, API call sequences, and how sensitive the code is. Tools created and studied before like AndroidAPIMiner [1] use a supervised learning algorithm to detect malware by extracting the frequency of API calls from applications. They concentrate on permission-related API calls and use balanced API dependence graphs to analyze the behavior of applications. These graphs are used as features, and a machine learning algorithm is used to introduce highly accurate malware detection. AndroidAPIMiner uses machine learning to classify the behavior of apps using sequences of threat modalities, then uses these modalities as features to enforce efficient malware identification and family classification. On other hand, tools like AndroSimilar [10] are based only on signature-based detection, which is used to identify only recognized malware samples. Its primary function is to collect, remove, and analyze malware-affected Android APKs. In general, all these tools specify how apps behave when security-sensitive procedures, such as permission-protected methodologies, source and sink techniques and dynamic code loading methods, are invoked. They separate contexts to illustrate the purposes of security-sensitive activities, then use these contexts as features to enforce precise malware detection.

The second method that was studied employs dynamic analysis, which attempts to solve the static analysis' shortcomings. The conduct of the application in a real-time environment is observed using this approach. This method examines the application's output when it is running, effectively resolving the problem of dynamic code loading in static analysis. Also, Android apps with obfuscated or encrypted source code may be handled by dynamic analysis. TaintDroid [32] is a complex taint analysis framework that detects confidential data leakage in Android apps.

It can now perform system-wide taint monitoring and report information leakage in applications without false positives after modifying the Android virtual machine interpreter. Tools like CrowDroid [13] recognize malware by initiating and running device calls. They identify malware by watching and reflecting on the actions of events in an Android app or at runtime. Typically, they create signatures by capturing inputs and outputs depending on the application's functionality. Furthermore, emulation-based detection is used by DroidScope [33] It has features that are dependent on the OS's review and Dalvik Semantics. In the sandbox, it does a separate analysis by dealing with the class.dex file, which is broken down into an easily readable format.

Nevertheless, since battery-operated systems use a lot of energy, complex analysis is not possible. When a valid application lets more device calls, anomaly-based identification strategies waste time and power, and they provide false findings. Taint analysis does not monitor control flow, whereas emulation-based monitoring covers only a small area and ignores new malware. So, techniques that involve a kind of hybrid analysis was required for a better research. For example, with Andrubis [20], the output of a static analysis that is retrieved by inspecting byte-code and the AndroidManifest.xml format, is fed into a dynamic analysis, which then performs process tracing and device level analysis while the program is being executed.

What's more, machine learning methods, which, when paired with program analysis techniques, can instantly predict behavior characteristics of applications, have become prevalent in the detection of malicious software, with both static and dynamic approaches. ML is based on Artificial Intelligence, which helps the machine to learn and evolve without having to be directly programmed. Not only in computer science, but also in a wide range of applications such as atomic physics, machine learning has proven to be successful. Although machine learning is incorporated in many tools already mentioned, another great example of ML combined with hybrid analysis is SAMADroid [28]. It's a three-level hybrid malware detection method for Android that's a reliable and effective malware detection approach. However, since it is reliant on server contact, the malicious activity of Android APKs is observed at a remote location. Also, Crowdroid [13] blends supervised learning algorithms to introduce efficient malware identification by performing complex analysis to track runtime activities such as API calls, device calls, and unseen icon activity.

More specifically, this work was mostly inspired by two other research projects: EnDroid [23] and Omnidroid [18]. For EnDroid, a novel dynamic analysis scheme is applied, that uses ensemble learning to define and distinguish malicious and benign applications. Ensemble learning improves the classification accuracy of base learning algorithms by integrating data from several base machine learning algorithms. To determine which base classifiers are effective and which are not, EnDroid uses Stacking. By merging templates constructed from various base classifiers with a meta-classifier, stacking typically achieves the best generalization accuracy. The complex analysis behind it includes system-level action tracing as well as general application-level malicious activities such as data theft or malicious data interaction. The researchers there, use a variety of machine learning algorithms to test EnDroid's efficacy, and found that stacking current ensemble approaches achieves the best results. Via a series of tests, they confirm EnDroid's efficacy in detecting Android malware and classifying it into families. They also equate EnDroid's malware detection efficiency to that of Maline, a cutting-edge dynamic analysis platform, to demonstrate its supremacy in malware detection. In addition, they use the chisquare feature filtering algorithm to filter out obsolete or noisy features and extract critical ones. In real-world implementations, these crucial features aid in the detection of dangerous actions.

Regarding Omnidroid project (that is also mentioned in a later section thoroughly), it is a detailed collection of dynamic and static features from Android apps. Other researchers will use this analysis to enhance and create new automated malware detection strategies for Android devices. At the same time, the features of this dataset make it ideal for use as a benchmark dataset for experimenting with and testing various algorithms and techniques. All of the data in the OmniDroid dataset is supported in JSON and CSV formats and is freely accessible to make it easier to use. AndroPyTool, an automated open source platform for dynamic and static review of Android software, was used to build the OmniDroid dataset. The efficiency of many state-of-the-art ensemble classifiers was investigated in order to determine the viability of using the features chosen to build special identification or classification models using ensemble techniques. It also shows the benefits of using the platform and dataset to create ensemble methods for detecting and classifying Android malware. Ultimately, an Android malware identification strategy is introduced, which is built on the integration of static and dynamic features through the use of an ensemble of classifiers with a voting scheme.

3. Problem statement

The fundamental problem of this master's thesis will be introduced in this section, plus an objective and the requirements will be described. The primary research issue will also be enhanced. Software applications or libraries in general are continually changing and so are malware and Android malware as well. For software devices, new features and fixes for bugs or security flaws are also implemented. Hence, in order to create new cyber threats, malware that exploits security flaws, has to always modify its application code. This leads in a battle between ensuring stability and spreading malware efficiently. Particularly, code modifications inside the operating system or its applications, potentially create new flaws that can be abused by more recent malware models. In this context, malware is continually evolving, but the security measures and detection systems have to do the same as well.

This thesis attempts to highlight the question of Android malware identification through the use of machine learning methods from various viewpoints. It is therefore meant to analyze the effectiveness of these approaches when used to address the problem illustrated, including requisite methods to accomplish this conjunction and to build innovative methods aimed at reliably identifying and quantifying malware. In particular, such a study is based on machine learning classifiers, supervised models being used to predict the class of new unlabeled examples after a training phase from defined samples. Here, such techniques are applied to tackle two separate challenges. As malware identification models, machine learning classifiers may be used to characterize malicious samples into two groups, those being malware or benign applications. As family classification methods, they are sometimes used to evaluate the malicious category of applications already identified as threats.

On top of that, the estimation accuracy for more modern malware can be impaired by testing a machine learning model with obsolete malware. New malware techniques and their essential functions are never learned, so the machine learning algorithm remains unaware of their unique attributes. Also, in order to estimate the identification of untrained Android malware, the core issue is that many scientific papers, as well as more recent ones, use old malware in supervised machine learning approaches, which creates new problems on its own. So, another aim of this thesis is to construct and test a range of new malicious and benign applications. It would still be accessed if older malware is present at the same time, as it may be useful for additional work.

The core study challenge runs as, how accurate is the quantitative detection of recent Android malware by analyzing multiple distinct approaches to hybrid analysis in combination with ensemble learning. In the assessment, the projected quality of each will be measured and the thesis will be concluded by addressing the underlying subject of the study, including a summary and possible future activities.

Engineering these methods for machine learning models though, whether for identification or malware family classification, requires a sequence of steps, from gathering a diverse collection of samples to validating and checking the qualified methods. It is possible then to define a protocol involving the following steps.

1. Comprehensive selection of both benign applications and malware. Throughout the context of family grouping, examples ought to be obtained from multiple malware families and a dataset that contains mostly new and relevant apps.
2. A variety of features capable of defining the behavior of each program and rendering distinctions between malicious and benign signatures are derived by malware detection and other tools.
3. Introduction and implementation of a functional model for hybrid analysis of both static and dynamic analysis with all kinds of features.
4. Multiple different samples defined as objects holding the extracted features, are given to the chosen machine learning algorithms in order to train them.
5. Appliance of feature engineering and optimization on behalf of the trained algorithms for any performance escalation possible.
6. Model testing and evaluation with proper machine learning and statistics metrics.

Finally, this thesis was organized across multiple goals in order to better understand the need for machine learning methods to address activities like malware. Some of them are, to analyze the most significant behavioral characteristics of android malware that must be taken into account when developing software for Android malware identification, to build the appropriate tools for automatic hybrid feature extraction and test efficiency or time consumption. The observations of these objectives, approach, implementation and results are outlined in the following chapters.

4. Approach

The approach method employed for this work, will be described in this segment, including the measures taken to decide the right data sets to be chosen, classifying the android applications as benign or malware and the parameters used to select the most effective resources for this work. Eventually, an elevated overview of a model of machine learning and testing will be presented.

In order to minimize the impact of emerging Android malware, these problems already mentioned, call for new and more efficient detection approaches. Therefore, here, it is proposed a method for early detection of Android malware by parallel machine learning classifiers that use various algorithms with behaviors that are clearly distinct. In the learning stage of the model, a variety of static app functions are used. In order to identify a given new application, the trained models are mixed using different mixture schemes to create a composite model that generates a verdict of malware or benign results. The primary contributions to this study are the upholding: Using concurrent ensemble machine learning classifiers, specifically stacked architecture used distinctly on 2 different groups of features (dynamic and static) combined with a final averaging function like a voting classifier, balancing individual weaknesses, a new Android malware detection technique is developed with a different structure than previous works.

A different approach is also introduced for detecting Android malware based on a extensive feature (different approach on features and permissions, mentioned later in 5.1.) system that reflects the distinction between hostile applications and friendly applications as machine learning functionality, including the hazardous permission details used for the first time in the components.

Furthermore, the purpose of the process described in this section is to demonstrate the feasibility and ease of use of the dataset when using it to build classifiers through machine learning methods (in particular, ensemble methods. In other words, show that the dataset that is being used for testing purposes, is usable out-of-the-box, and although some promising experimental results are currently obtained, there is still a large room for improvement.

4.1. Dataset

Over the years, machine learning classifiers have played a great role in designing intelligent systems for various problems. Especially, in finding and detecting malware on all networks, ML techniques are gaining momentum. So, the proposed model is based on supervised machine learning, whereby a named dataset acquires the characteristics defined in the previous section and uses them to construct and train a model. Also, the aim of the method outlined in this section is to evaluate the viability and ease of use of the dataset by using it to create classifiers using machine learning techniques (in particular, ensemble techniques) instead of constructing a highly accurate malware classifier. In these terms, it indicates that the sample used is out-of-the-box accessible and there is still a wide space for progress, while some positive experimental findings are currently obtained. Such approaches were used independently over the collection of derived static and dynamic features, and essentially a fusion-based approach is proposed where all types of features are merged.

Statically derived features could be used to construct representative sample structures where each location reflects the frequency of occurrence in the sample that are present with a certain characteristic, i.e. the number of times invoked by a certain API call. Given a set of samples X of size n: X = {x1, x2, . . . , xn}, each sample xi is represented by a vector of m static features: xi = {sc1 i , sc2 i , . . . , scm i }. At the same moment, each xi sample is labeled according to its mark li , li ∈ {0, 1} as benevolent or malicious. Our task is to train the classifier that creates this relationship: Cls(xi) = (p(li), li). Then, the use of dynamic features derived during the execution of the sample in an emulator is evaluated until the static features have been analyzed. Each analysis provides a transient sequence of actions conducted during the execution, in which each action is connected to a group, for example, access to the file or filepath.

The proposed model utilizes static and dynamic analysis to retrieve essential components by decompiling programs and evaluating them using the ideal machine learning solution in order to extract features that can best represent application actions.

4.1.1. Features

There are actually a variety of resources to extract static and dynamic features from Android apps. Each, though, relies on a precise category of features, so it becomes impossible to achieve a full collection of both static and dynamic features. It requires setting up each tool with its respective configuration files and settings, as well as grouping the independently collected outcomes to construct a full data collection.

AndroPyTool is an interactive Python application intended to obtain different dynamic and static features from a collection of Android apps. It integrates the most used tools for Android malware detection, conducts source code inspection and retrieves activity information while the sample is run in a managed environment [18]. For any analyzed application, including a broad batch of fine-grained representative characteristics, the tool offers a comprehensive report. As seen in AndroPyTool documentation and relevant publication [18], every application file (namely apk) follows a pipeline comprising seven steps :

1. Filtering APK: The purpose of the first stage is to inspect using the AndroGuard tool for each sample to determine if it is a legitimate Android program for the study.
2. Virustotal analysis: A report is obtained for an application from the online web application Virustotal. The report includes the conclusions of the scan conducted by and resulting from Virustotal data from the application's review by more than 60 different anti-malware engines.
3. Dataset partitioning: In this stage, which is optional, if tagged this way by at least e antivirus based on VirusTotal study, each sample is classified as malware. The end user of the tool, who can set his/her own e-threshold, can change these parameters.
4. FlowDroid execution [4]: This method is run against each sample, based on taint analysis.
5. Processing of FlowDroid outcomes: There is a processing step of those outcomes provided by FlowDroid to extract links. The cause for splitting the extraction and processing of information flows lies in the possibility of different representations, thus enabling the processing step to be independently altered.
6. Execution of DroidBox: A customized version of DroidBox, a dynamic analysis tool for Android that includes the Strace tool, is run against the data.

7. Feature extraction: The tool captures, reorganizes and structures the findings as records of derived features in this process. For both implementations, the cumulative dataset of extracted features is given in the following formats: as a comma-separated value or as a JavaScript Object Notation file.

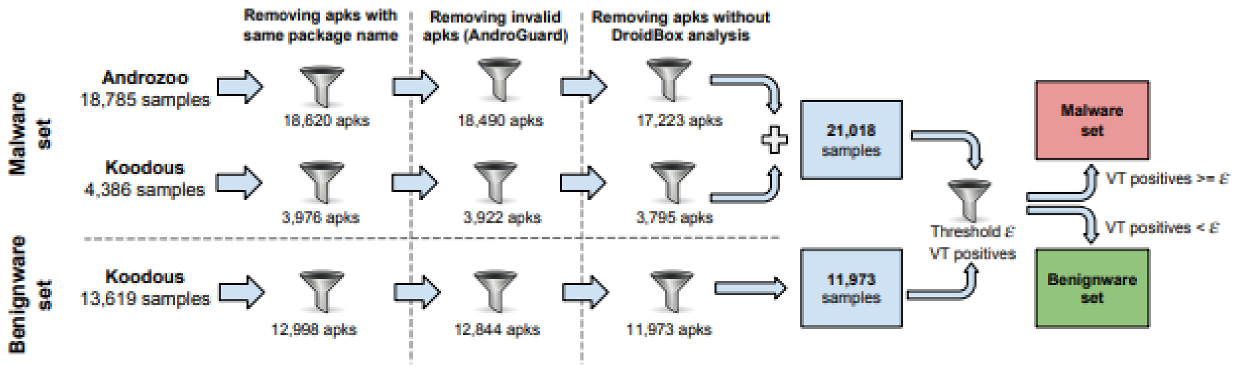


Figure 4.1. OmniDroid dataset [18]

AndroPyTool was also used to create the OmniDroid dataset [18], the primary dataset used for our testing purposes. A large collection of samples from two separate sites is analysed using this method. At the very beginning, from a dataset of 100,000 samples that Koodous gave to OmniDroid for research reasons, both benign and malicious samples were obtained. To complement the first set and also to encourage diversity inside the malware set, additional samples from the AndroZoo portal have been included, while avoiding alternative causes of prejudice in between the samples. In order to exclude repetitive applications and those found to be invalid, a filtering approach was followed. The first filter consists of eliminating samples represented by repeated names of packages, thus avoiding several instances of the same program and its associated vector features. Then, with the AndroGuard method considered part of AndroPyTool, invalid apps that could not be run were found in order to delete them. The third stage pursues the same target, but removing samples in this case that could not possibly be implemented in the Android emulator used by DroidBox.

AndroPyTool has actually run on multiple sample sets until a large amount of samples have been analyzed: 21,018 samples of malware and 11,973 benign samples [18]. However a distinguish between malicious and non-malicious samples is already made by the Koodous dataset, all samples were sent to VirusTotal. This helps each app to receive a modified report that contains the outcome of the scan collected from a collection of antivirus engines that can be used to mark each sample as malware or benign. Therefore, for each sample, the rate of positives given by the antivirus engines deployed by the VirusTotal portal is included as pre-static information. The final distribution of samples to a malware or benignware range would be in the hands of the OmniDroid dataset developers, who will set their own parameters, because a low rate of antivirus reporting malicious material may be attributed to false positives. A method such as the one proposed by AndroPyTool can be used in this line, where even the classification of each sample is set according to a given E-threshold [18].

The finally constructed dataset comprises a subset of the collected applications because of the heavy computing load needed to extract the complete analysis from each sample. To retain a healthy dataset containing both malware and benign samples, this decline was discussed. The threshold parameter E was set to 1 for that reason. As per this criterion, 11,000 malware and 11,000 benignware samples are composed of the finally created and released dataset.

All of the features available in this dataset were obtained by running AndroPyTool from the above collection of processes mentioned. A basic understanding of the amount of extracted features can be seen here, where some of the most significant feature categories are listed. Through further analysis in [18], only a subgroup (at least 80%) of each feature category of the below list is used as features in our dataset.

• Permissions:	5,501
• Services:	4,365
• Opcodes:	224
• Receivers:	6,415
• API calls:	2,129
• API Packages:	212
• System commands:	103
• FlowDroid:	961
• Activities:	6,089

The large range of features found in the dataset, derived utilizing both a static and a dynamic approach to evaluation, allows the creation of resilient methods for identification and classification. While strategies that rely on a reduced collection of features can be fooled quite efficiently, those that use a wide set of features to define the actions of the application present a major resistance against this form of attack.

In this dataset, all the features derived describe a broad space from which various distinctions and considerations can be made. For example, when observing malware or safe samples, it is possible to examine the existence of unique features. The threshold e has been adjusted to 1 for such tests, so benign samples would be those that all antivirus engines deem as such, whereas those that are at least one antivirus classified as malware are added to the malware set.

Regarding pre-static analysis, this includes background information about the applications, such as the name of its file and checksums, the number of positive fields recorded in its VirusTotal analysis, and the total number of engines from which the results of the scan were collected. While several of these components can not necessarily be viewed as behavioral characteristics, these fields can be used for further research to keep track of the APK. This form of features also contains a sample categorization according to the AVClass tool, which attempts to find a consensus between the outputs generated by the numerous VirusTotal-operated antivirus engines [18]. Starting from a series of numbered samples, this tool extracts a token array, identifies aliases, applies several filters, and shows the most convenient token for each sample.

A collection of additional features, that are derived using static analysis methods, is retrieved from the samples and inserted into the dataset until the pre-static features are retrieved. These attributes are: the name of the package, permissions, opcodes, the name of the key operation, API calls, strings, device commands, and the collection of intents that can be individually handled by the activities, utilities, and receivers. The primary purpose of this set of features was to have an overview into the intended behavior of the program and the variety of actions it may involve on the basis of a static code analysis which does not mean code execution [18]. Although this method of analysis does not represent the sample's actual actions, which can only be disclosed once the sample is run, it feeds useful information to the analysis of a study. A detailed description of API calls contained in the code is given in the first place. It is essential to mention that this method of analysis does not identify certain functions that are triggered through reflection or dynamic code loading, among other evasion techniques. API calls can be found in the OmniDroid dataset, clustered by the category in which they are specified or by their package. Furthermore, permissions display system features that may be accessed by the app, and system commands give a description of behavior that can occur at a low level, such that unusual behavior, like privilege escalation, may be exposed. Some other static features include Dalvik opcodes that are produced by evaluating the Dalvik bytecode that are effective for low-level discernment of the behavior of the sample and about the same moment, a collection of intents used to invoke other elements of the application allows the sample to be profiled based on the actions done and the activities that affect those actions. Finally, it also contains a report assembled using the FlowDroid tool. In order to determine relations between a source and a drain, this is a valuable instrument that conducts taint analysis over the application code. This sources and sinks, previously defined by the SuSi framework [31], make it possible to model the data leaks carried out over the life cycle of the program.

The dynamic analysis data gathered with a modified version of the well known Droidbox system are grouped by this set of features. This architecture offers useful features, such as network behavior or accessed files collected during execution, that can be evaluated. To receive more detailed complex reports, this system was modified. This alteration includes deploying the Strace tool within the Android emulator application to collect a list of all system calls made at the Linux level during runtime. There were also two additional DroidBox adjustments. On the one hand, with the intention of stimulating the sample under examination with a larger amount of simulated user behaviors on the screen and buttons, the behavior of the MonkeyRunner tool already implemented in DroidBox has been modified. This updated version, on the other hand, allows the sample to be run in a non-GUI environment, allowing several simultaneous simulator instances in device nodes to be run [18]. The use of a dynamic analysis technique, in comparison to static analysis, makes it possible to interpret the actual actions displayed in a virtual environment where the software is performed. The OmniDroid dataset includes all the occurrences collected by the DroidBox tool and a log created with Strace for each analyzed application in the entire production log. These two types of knowledge include vast volumes of information that must be filtered and interpreted if used in accordance with the machine learning classifier.

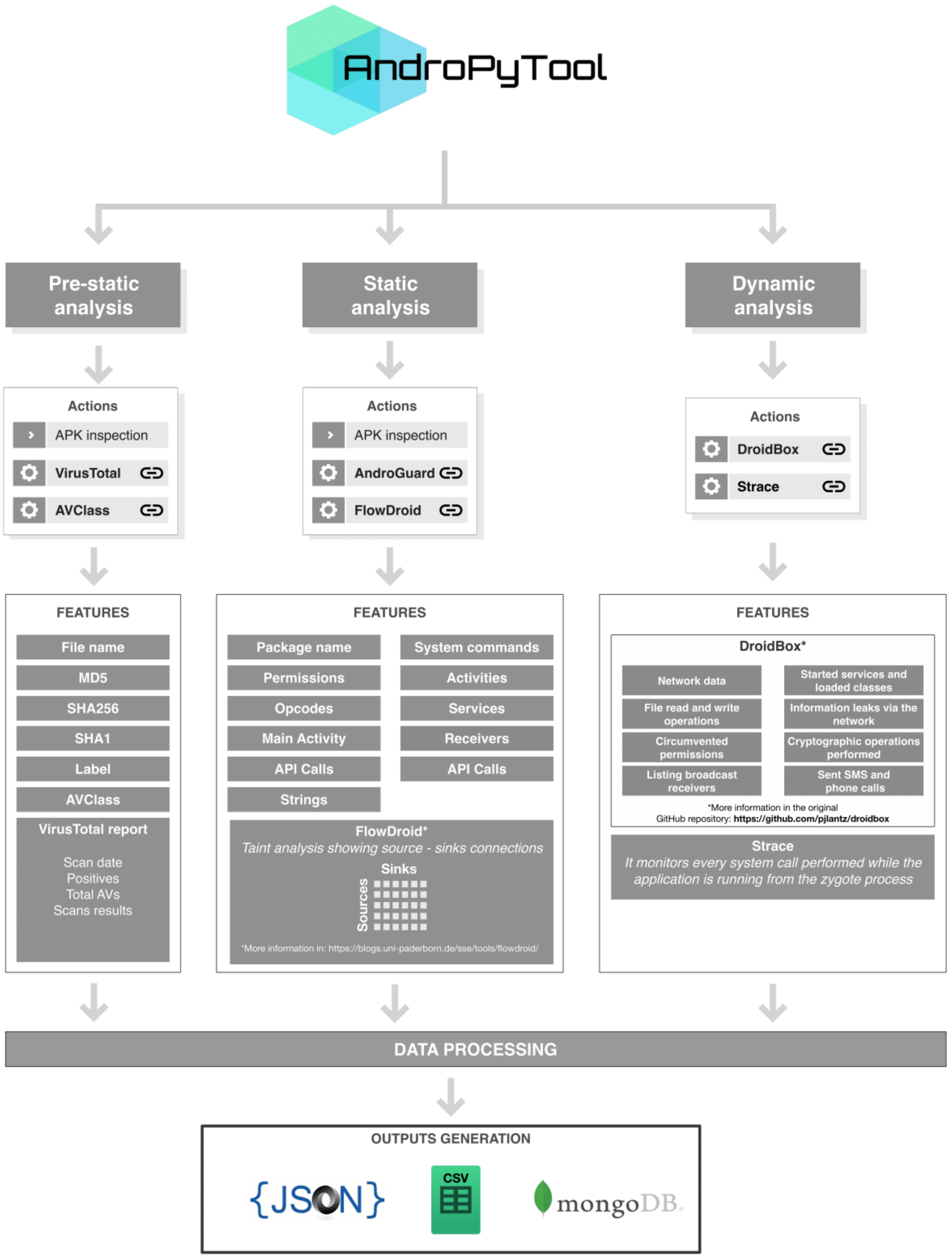


Figure 4.2. AndroidPyTool overview [18]

4.2. Model Architecture

These approaches (parallel classifiers) were used independently over the collection of derived static and dynamic features and essentially a fusion-based approach is proposed where all types of features are merged. To train and evaluate these algorithms, the same threshold related to the minimum number of positives specified in the previous segment, described by e = 1 (as mentioned later), is again used.

In general, four well-known state-of-the-art ensemble methods for classification were trained in parallel with different combinations of features. These algorithms, executed with the Scikit-learn library for Python, are the following: AdaBoost, Random Forest, ExtraTrees, Gradient Boosting (all of them using a parameter of 100 internal estimators. Their results are then fed on a final estimator, in this case, Logistic Regression, leading in a stacked model. This is repeated for both dynamic and static features independently, combining the results with a final Voting Classifier with multiple weight approaches (the reason behind the selection of these specific algorithms is explained in the next section).

When opposed to a dynamic-based approach, it has been shown that using a classifier with an input based on static features increases accuracy [18]. Despite this, the increasing sophistication of threats necessitates the use of all possible methods in order to determine the type of a suspect sample. To this purpose, a new method was invented based on the fusion of activity data derived from a hybrid analysis that merged static and dynamic elements. This combination method entails merging the best classification models with each category of function (static and dynamic) to create a voting classifier in which each feature category model contributes to the final categorization. So, the final classifier of this model uses both static features like API calls (which provide the best results in the static comparison) and dynamic features like the various representations evaluated in the dynamic analysis method. As a result, three vectors are generated by combining API call functionality with transfer probabilities, frequencies states, and a combination of both. In practice, the voting classifier's specification incorporates the results of two stacked classifiers that are responsible for classifying the input vector's static and dynamic features parts. As per two weight parameters W_1 and W_2, each classifier leads to the classification of a linear set of features.

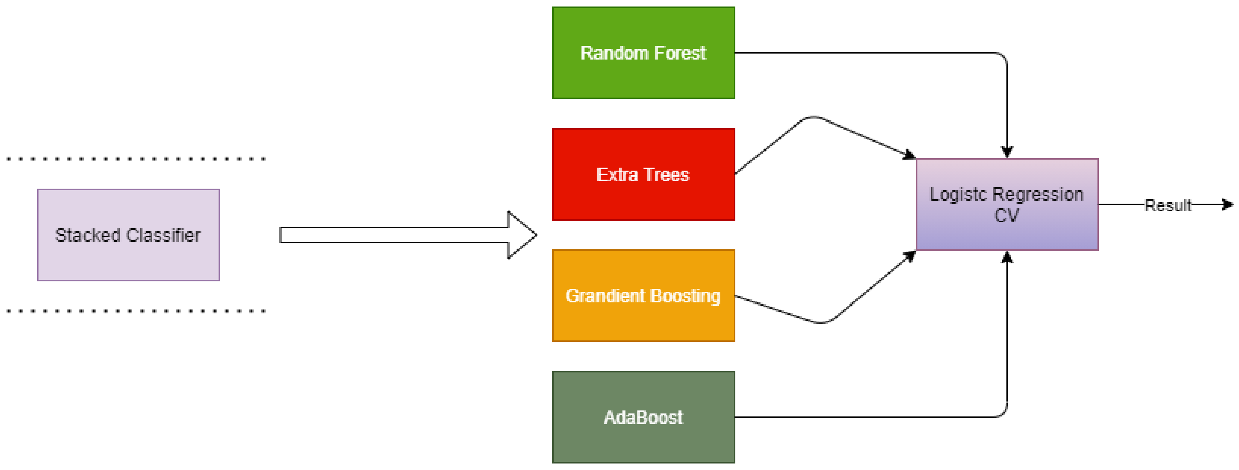


Figure 4.3. Model's Stacked Classifier

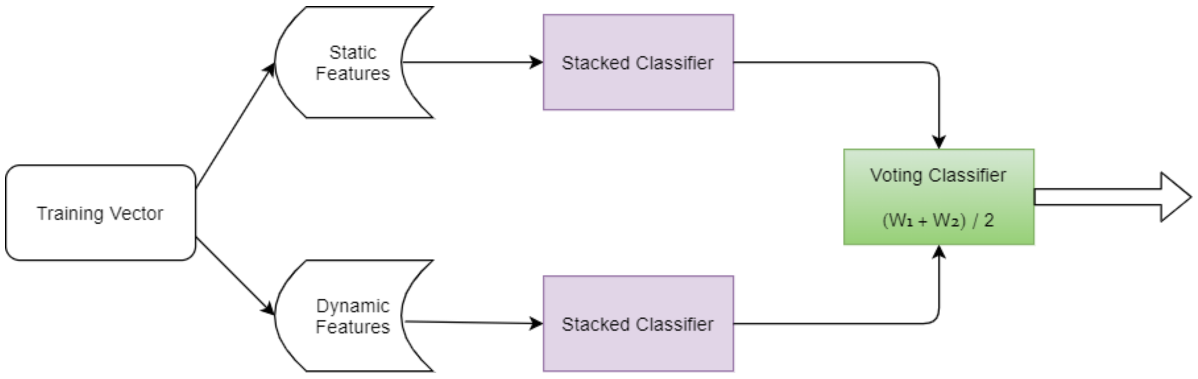


Figure 4.4. Model Architecture

4.2.1. Ensemble Classifiers

There are several classifiers that could be used in order to obtain a result that suits our purposes inside a stacked model, but instead there were chosen 4 ensemble methods. As already mentioned, ensemble methods aim to increase generalizability and robustness over a single estimator by combining the predictions of multiple base estimators constructed with a given learning algorithm. Typically, two types of ensemble approaches are distinguished: The guiding force behind averaging methods is to construct many estimators independently and then average their forecasts. Since its uncertainty is minimized, the combined estimator is normally better than either of the single basis estimators. In boosting methods, on the other hand, base estimators are constructed sequentially and the cumulative estimator's bias is reduced. The aim is to create a strong ensemble by combining many poor models. In order to build our basic estimator, 4 ensemble methods are used. These are AdaBoost, RandomForest, ExtraTrees and GradientBoosting. The reason behind these specific algorithms, is first, the fact that they are 4 of the most known and usable ensemble methods in Android malware detection in general and secondly they are a great example to compare final statistics and results as shown in other research works.

4.2.2. Stacked Approach

As far as our main estimator goes, a stacked approach is used. Stacking is the process of teaching a learning algorithm to merge the assumptions of several learning algorithms. Every one of the algorithms are trained with the available data first, and then a fusion algorithm is taught using all of the other algorithms' predictions as additional information to produce a final prediction. Stacking can potentially represent all of the other ensemble methods if an arbitrary combiner algorithm is used, but in reality, a logistic regression model is often used as the combiner. In most cases, stacking outperforms all of the qualified models individually. It has been effectively implemented on both supervised and unsupervised learning functions (density estimation) [31].

Stacking answers the issue of how to pick between various machine learning models that are skilled at solving a problem in different ways. Another machine learning algorithm that knows when to use each model in the ensemble is used to address this problem as a final estimator [31]. In contrast to bagging, the models in stacking are usually different and work on the same dataset rather than samples from the testing dataset. Also, unlike boosting, stacking utilizes a specific template to learn how to better integrate the observations from the participating models, rather than a series of algorithms that correct prior models' predictions.

A stacking model's structure consists of multiple base models, also known as level-0 models, meaning models that fit on the training dataset, and a meta-model that incorporates the projections of the base models, also known as a level-1 model, that learns how to best mix the previous predictions [31]. The meta-model is educated using out-of-sample data projections created by base models. That is, non-training data is fed into the base models, predictions are made, and these estimates, along with the predicted outcomes, form the input and output combinations of the training dataset used to match the meta-model. In the case of regression, the results from the base models for use as feedback to the meta-model could be actual numbers, likelihood values; in the case of classification, the outputs from the base models may be probability values, or class labels.

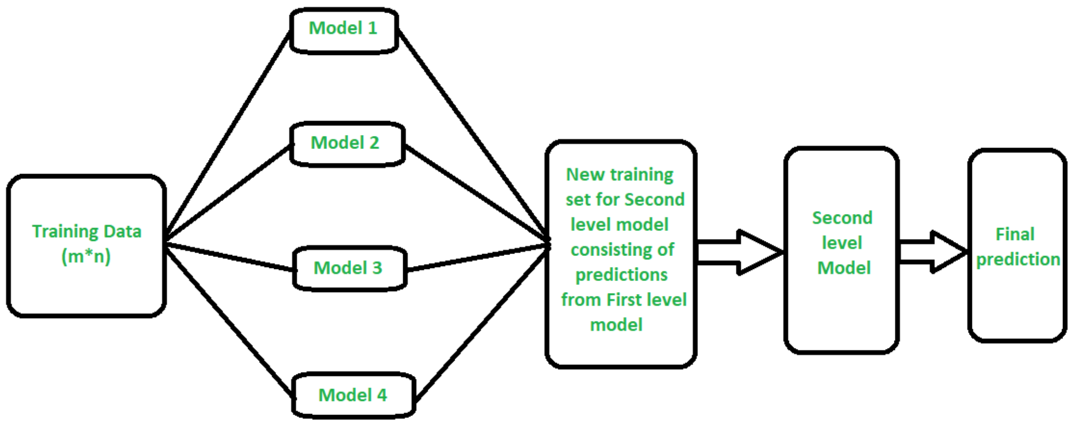

Model 1
Model 2
Training Data (m*n)
Model 3
Model 4
New training set for Second level model consisting of predictions from First level model
Second level Model
Final prediction

Figure 4.5. Stacked ensemble learning

The most popular method for planning the training dataset for the meta-model is k-fold cross-validation of the base models, with the out-of-fold projections serving as the foundation for the meta-training model's dataset. The inputs to the base models, such as input elements of the training data, can also be used in the meta-training model's data. This will give the meta-model more context in terms of how to better combine the meta-forecasts. The meta-model can be learned in turn on this dataset until the training dataset for the meta-model has been fitted, and the base-models can be trained on the entire initial training dataset until the training dataset for the meta-model has been prepared. When several distinct machine learning models have knowledge on a dataset, but in varied contexts, stacking is suitable. Another way to put it is that the models' observations or flaws in predictions are mutually independent or have a poor correlation.

Base models are often dynamic and varied. As a result, it is always a great idea to use a variety of models, such as linear models, decision trees, neural networks, and others, that make quite reasonable interpretations about how to handle the predictive analytics problem. The meta-model is sometimes straightforward, allowing for a seamless analysis of the base models' observations. As a result, linear models, such as linear regression for regression tasks and logistic regression for classification tasks, are frequently used as the meta-model. This is a standard practice, but it is not always needed.

Since a basic linear model is used as the meta-model, stacking is often referred to as "blending" [31]. A weighted average or mixing of the estimates provided by the base models is used in the analysis. Stacking is intended to increase modeling efficiency, but it is not expected to do so in all situations. The nature of the situation and whether it is fairly well described by the training data and complex enough that there is something to gain by integrating projections determine whether or not output can be improved.

4.2.3. Voting-based fusion

The stacked model described above is used separately in different static and dynamic feature groups. The final estimation of the sample being malware or benign is performed through a special voting classifier, fusing the results from the previous feature groups, using weighted approach.

Classifier fusion is based on the assumption that all classifiers in a given set are competing rather than complementary. Each factor contributes to the classification of an input vector. In the basic voting case, the selection is done depending on the amount of votes provided to each of the classes by the individual classifiers, with x being assigned to the class with the most votes [22]. When dealing of data sets with more than two classes, relations between certain classes are very common in the final decision. Several conditions should be considered in order to solve this dilemma. To take the decision at random, for example, or to use an alternative classifier whose end aim is to skew the decision against a certain class. This assertion, though, is not necessarily true. For example, simple voting may perform worse than all of its members in certain situations. As a result, using a weighting approach as a means of partly overcoming these problems has been suggested. The error rate associated with the basic voting system and its individual components is a significant problem that has piqued the interest of many researchers. It has been demonstrated that if any one of the classification methods being integrated has an error rate of less than 50%, the ensemble's performance would increase as more parameters are applied to the method.

Fusing results with voting could be made more resilient to the number of individual classifiers by using a weighted voting system. There are two general approaches to weighting that can be mentioned: Classifier weighting can be dynamic or static. The weights allocated to the specific classifiers will vary for each input vector in the operational process using a dynamic approach. The weights for each classification model are calculated in the training process of the static approach, and they do not adjust during the classification of the input patterns. The latter is the one used in our research scenario.

Specifically, we use a methodology that can be used to increase model consistency, with the aim of outperforming any particular model used individually for static and dynamic feature detection. The projections from different models are combined in a soft-voting ensemble. It may be used for regression or classification.

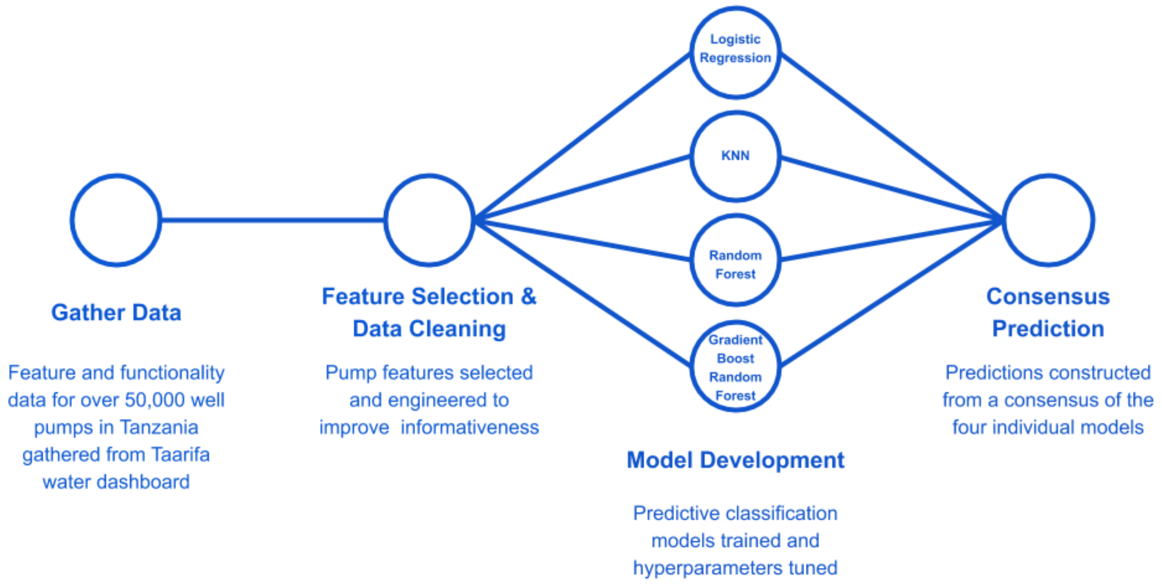

Logistic Regression
KNN
Random Forest
Gradient Boost Random Forest
Gather Data
Feature and functionality data for over 50,000 well pumps in Tanzania gathered from Taarifa water dashboard
Feature Selection & Data Cleaning
Pump features selected and engineered to improve informativeness
Model Development
Predictive classification models trained and hyperparameters tuned
Consensus Prediction
Predictions constructed from a consensus of the four individual models

Figure 4.6. Soft-voting model

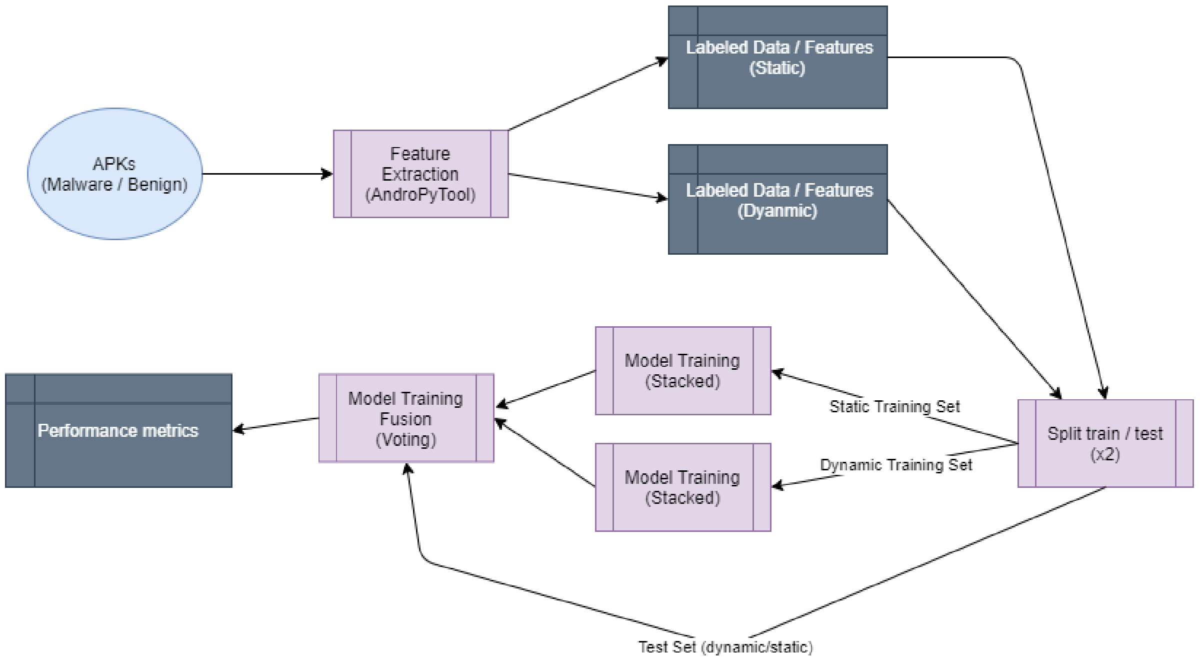

APKs (Malware / Benign)
Feature Extraction (AndroPyTool)
Labeled Data / Features (Static)
Labeled Data / Features (Dyanmic)
Split train / test (x2)
Static Training Set
Dynamic Training Set
Model Training (Stacked)
Model Training (Stacked)
Model Training Fusion (Voting)
Performance metrics
Test Set (dynamic/static)

Figure 4.7. Model as-a-whole

5. Implementation

This section includes a more detailed discussion and explanation of the proposed architecture. Any part's applicable code will be presented and clarified. A comprehensive overview of the applied code's components, including features, classes and in general of the way all the models were built together. Some information about the libraries and modules that were used will be given in the code sections where they were used. In conclusion, this chapter shows the complete way that tests were run and how the results were obtained, in order to have a better understanding of the benefits and drawbacks that the proposed architecture has to offer.

5.1. Workspace

Beginning with the nature of the architecture and the tests that were run, the programming language used, was Python (version 3.9), a scripting language. Its programming philosophy prioritizes code readability, as shown by the extensive use of indentation, while most of its constructs and object-oriented style are aimed at assisting programmers in writing simple, logical code for projects, independently of their size. Python was designed to be extremely scalable with modules, rather than having all of its features incorporated into its core. Because of its lightweight modularity, it's especially common for connecting programmable interfaces to existing applications. A minimal core language with a wide standard library and an interpreter that can be quickly extended. With libraries including TensorFlow, Keras, Pytorch, and Scikit-learn, Python is widely used in artificial intelligence and machine learning projects like this one, while it is also a very common language used in information security areas like exploit development. The platform used to create any necessary scripts and equally run it, is called PyCharm, a popular Python IDE through which any library needed was installed.

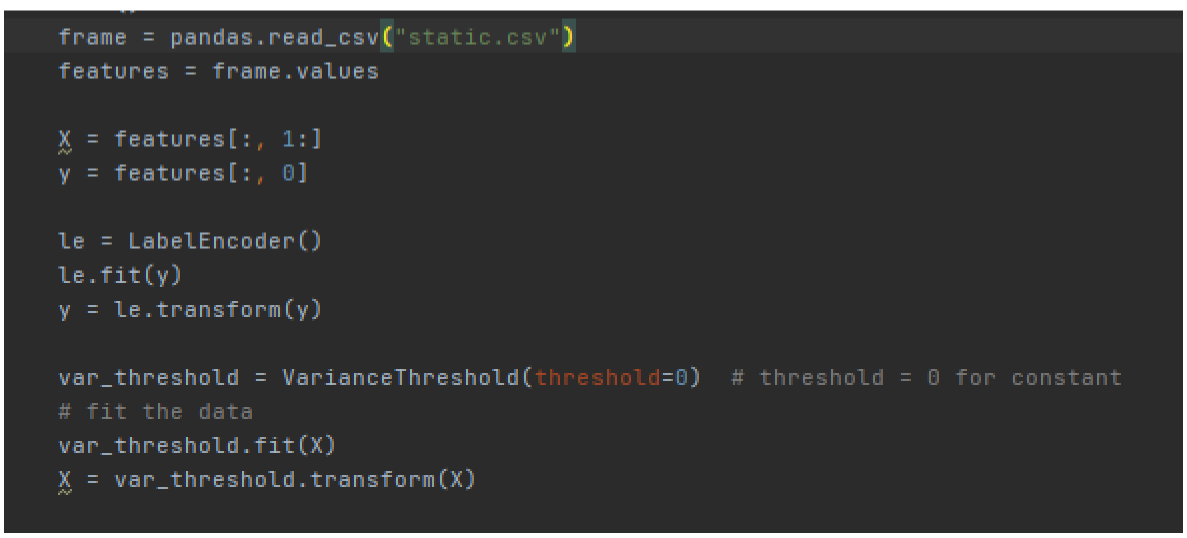
Regarding the libraries used for our purposes, Pandas [11] is one of the two libraries utilized. This is a data manipulation and interpretation software library written in Python. It includes data structures and operations for controlling numerical tables and time series, in general. The term panel data comes from econometrics which refers to data sets that contain measurements from different time intervals. Pandas is primarily used to analyze data. It supports data import from a variety of file formats, including comma-separated values and JSON. Pandas supports a wide range of data manipulation processes, including reconfiguring, picking and metadata management. Many of its useful features needed are the following: DataFrame objects with optimized aggregation for data processing, reading and writing content between in-memory data structures and various file formats, alignment and management of lost data in a unified manner and the possibility of reshaping and pivoting data sets.

The second and most important library used in our project, is called Scikit-learn, also known as sklearn. It is a Python machine learning library that is available for free. Support vector machines, random forests, and k-means are among the classification, regression, and clustering algorithms used, and it is designed to work with Python computational and science libraries such as NumPy.

5.2. Methodology

The first thing to be done after having essentially used AndroPyTool in order to have the datasets ready as csv files (explained earlier), is to load the datasets. CSV files are the most popular format for machine learning results, while they can be loaded in Python in a variety of ways. Generally, the module reader() in the Python API can be used to load CSV files and after the data has been loaded, it is converted to a NumPy array and used for machine learning. In our case, Pandas and the pandas.read_csv() function are used to load the datasets. This feature is extremely versatile, and it is probably the preferred method for loading machine learning data. The function returns a DataFrame from which we can begin summarizing and analyzing data right away. This process is done twice for static and dynamic features, independently.

Right after loading the dataframes, we retrieve the features (static and dynamic) into an array called X and their labels in an array we call Y. Using LabelEncoder() from sklearn.preprocessing, both input and output parameters in machine learning algorithms must be numerical. This implies that if the data is categorical, you'll need to convert it to numerical before fitting and evaluating a model. When dealing with categorical data for machine learning algorithms, encoding is a mandatory preprocessing phase. So, with LabelEncoder() target labels with a value between 0 and n_classes - 1 should be encoded. Instead of encoding the input X, this transformer can be used to encrypt the target values such as Y.



```
frame = pandas.read_csv("static.csv")
features = frame.values

X = features[:, 1:]
y = features[:, 0]

le = LabelEncoder()
le.fit(y)
y = le.transform(y)

var_threshold = VarianceThreshold(threshold=0)  # threshold = 0 for constant
# fit the data
var_threshold.fit(X)
X = var_threshold.transform(X)
```

Figure 5.1. Loading and processing data (python)

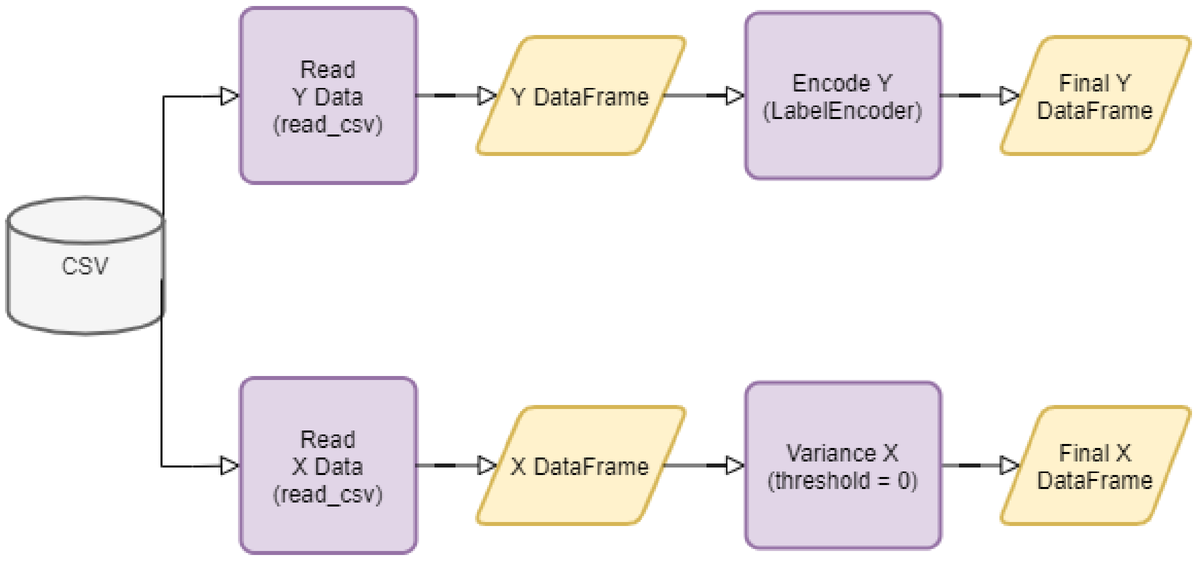
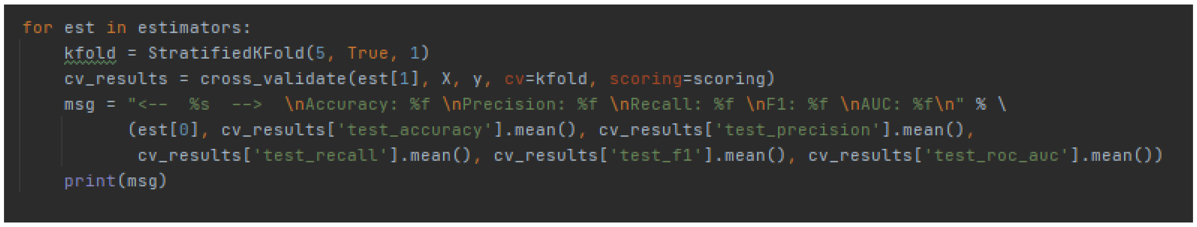


Figure 5.2. Loading and processing data

As soon as the encoding is completed, we go through a loop to test all of the estimators capabilities. The estimators tested here, taken directly from scikit-learn are the following: RandomForestClassifier(), GradientBoostingClassifier(), ExtraTreesClassifier(), AdaBoostClassifier(), VotingClassifier() and StackingClassifier with LogisticRegressionCV as final estimator. The final estimation considers the VotingClassifier() as a combining function between a dynamic and static feature assessment.



```python
for est in estimators:
    kfold = StratifiedKFold(5, True, 1)
    cv_results = cross_validate(est[1], X, y, cv=kfold, scoring=scoring)
    msg = "<--  %s  -->  \nAccuracy: %f \nPrecision: %f \nRecall: %f \nF1: %f \nAUC: %f\n" % \
          (est[0], cv_results['test_accuracy'].mean(), cv_results['test_precision'].mean(),
           cv_results['test_recall'].mean(), cv_results['test_f1'].mean(), cv_results['test_roc_auc'].mean())
    print(msg)
```

Figure 5.3. Testing estimators loop (python)

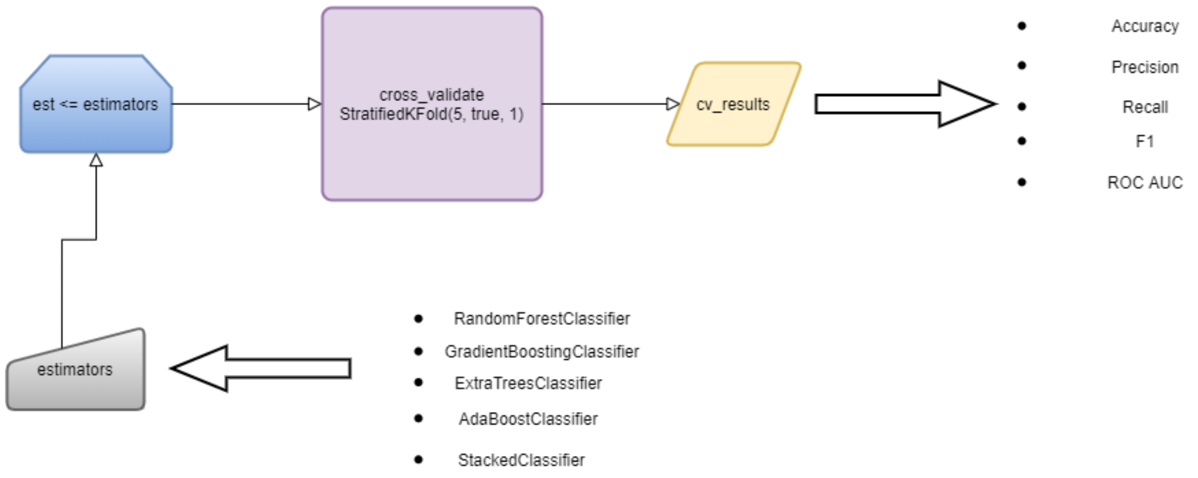


Figure 5.4. Testing estimators loop

It is highly recommended here to pay attention to a specific method used in order to have correct results, called cross validation. A conceptual error, is studying the properties of a prediction function and evaluating it on the same information: a system that simply repeats the labels of the samples it has just seen will have a top score but struggle to determine something meaningful on unseen data. Overfitting is the term used to describe these circumstances. To stop this, it is standard practice to set aside a share of the sample data as a test set X test, y test while running a machine learning test. The train test split assistant component in scikit-learn will easily compute an arbitrary division into training and test sets. There is also a chance of overfitting on the test range when assessing various configurations for estimators since the variables can be adjusted before the estimator works ideally. This allows information about the test set to accumulate into the model. To address this issue, another portion of the dataset may be set aside as a validation set, with testing taking place on the training set, followed by assessment on the validation set, and finally, when the test appears to be satisfactory, final evaluation on the test set. Even so, by dividing the accessible data into three groups, we greatly limit the amount of samples that can be used to train the model, and the outcomes can be influenced by a random pair of set selection. Cross-validation is a technique that can be used to solve this case. The validation collection is no longer required, but a test set should be kept on hand for final assessment. The training set is divided into k complete subsets in the simple technique, known as k-fold CV. For each of the k folds, the protocol is as follows: A model is created using the spreads as training data, and the model is then tested using the collected information. The average of the values computed in the loop becomes the output metric stated by k-fold cross-validation.

StratifiedKFold as a specific example that's being used, is a k-fold variant that generates stratified folds, with each set containing about the same proportion of samples from each class label as the entire set. Finally, with concern to the way the results are being processed and evaluated, these methods (that are explained in the next chapter thoroughly) are used through scikit-learn:

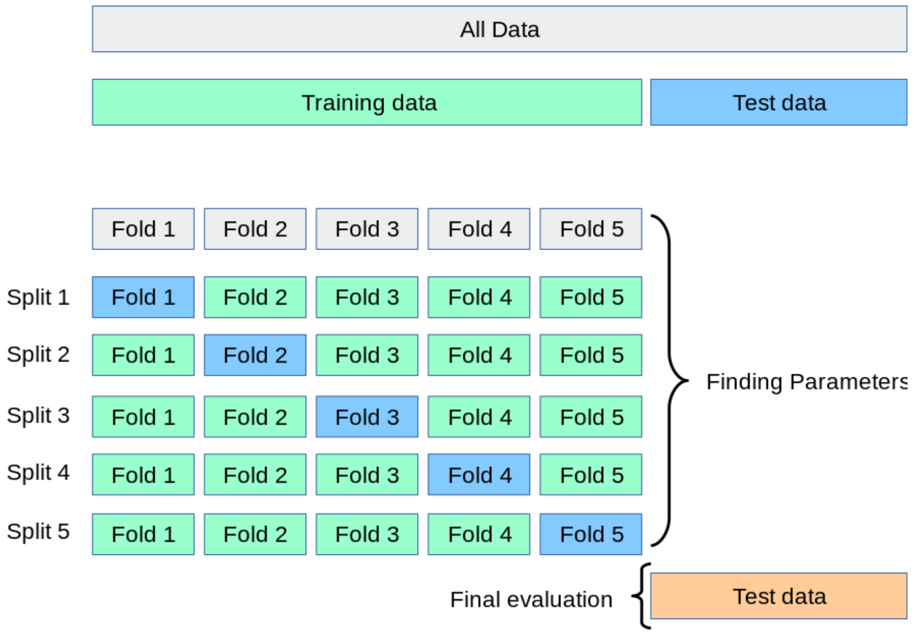


Figure 5.5. Cross-validation

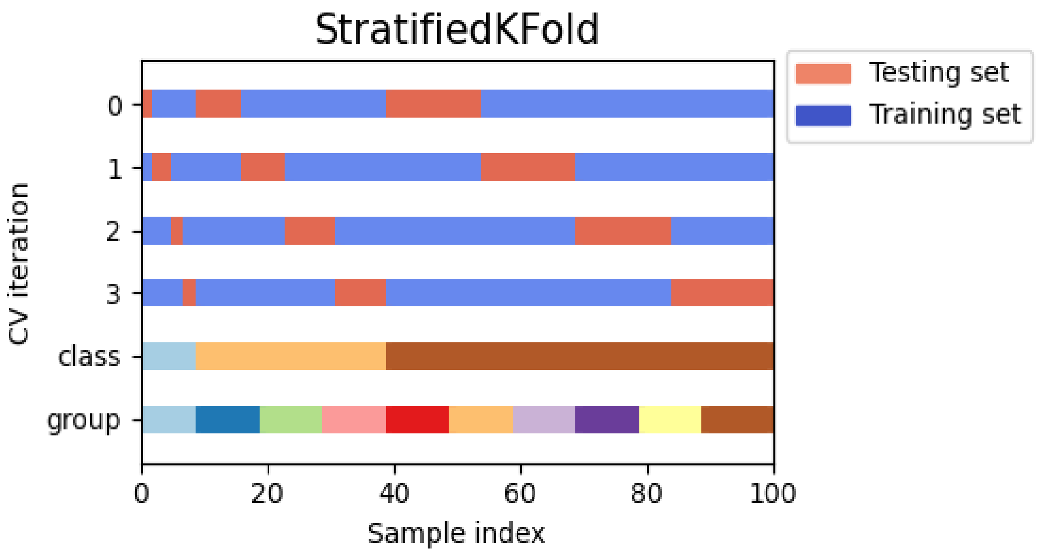


Figure 5.6. Stratified K-Fold

6. Results

The outcomes of the previously mentioned implementation and testing will be summarized in this chapter. For each function retriever and classification algorithm, there are performance metrics for particular features-classifier combinations and results obtained in general. In our case, we didn’t have to go further with testing and tuning different hyperparameters manually, since there was a different overall goal. The matrix is subjected to the 5-fold cross validation procedure in order to assess the accuracy of the classifier construct. As a result, the dataset is divided into five equivalent bits with no overlaps. Each phase of the assessment uses test data from one partition and a qualified model from the other four bits. The outcomes are combined to provide the classifier's final output results. The k-fold cross validation procedure is a widely used machine learning assessment tool that is well suited to our aim of assessing the overall effectiveness of classifiers in detecting abnormal malicious applications, which is simulated by the non-overlapping testing segments.

6.1. Metrics

- Accuracy
- Precision
- Recall
- F1
- AUC

Before diving into the explanation of these metrics and starting with another important concept, the confusion matrix, it is the most straightforward way to assess the success of a classification problem with two or more types of output. A confusion matrix is just a two-dimensional table where all dimensions include True Positives, True Negatives, False Positives, and False Negatives. True positives are the ones when the actual class of the data point was True and the predicted is also True. True negatives are the cases when the actual class of the data point was False and the predicted is also False. False positives are the cases when the actual class of the data point was False and the predicted is True while False negatives are the ones with True actual class and False predicted.

Regarding accuracy now, it is the most common efficiency metric. It's the number of accurate predictions divided by the total number of predictions. The amount of accurate predictions divided by the total number of input data is the ratio of accuracy. Here, we use accuracy_score [25] from sklearn in order to compute accuracy. Precision, on the other hand, is the percentage of correct records retrieved by a model, while the general amount of positives produced by the model is known as recall. The modules being used from sklearn.metrics are precision_score and recall_score [25].

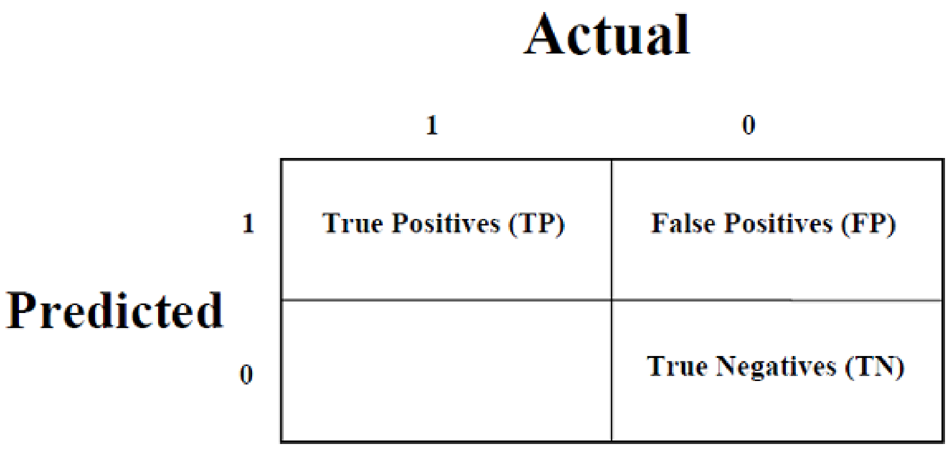
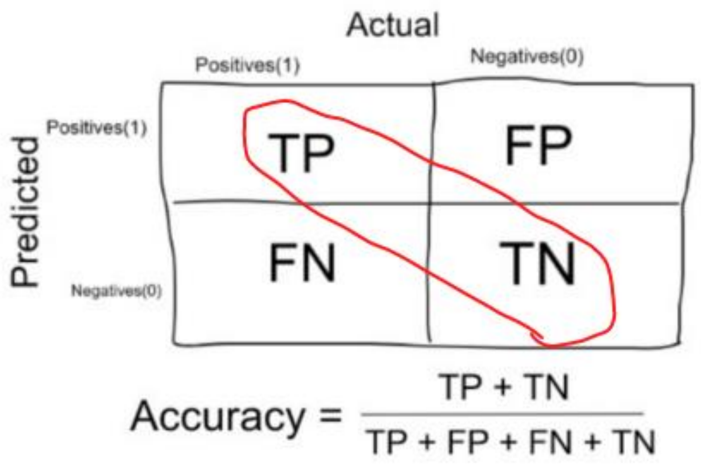
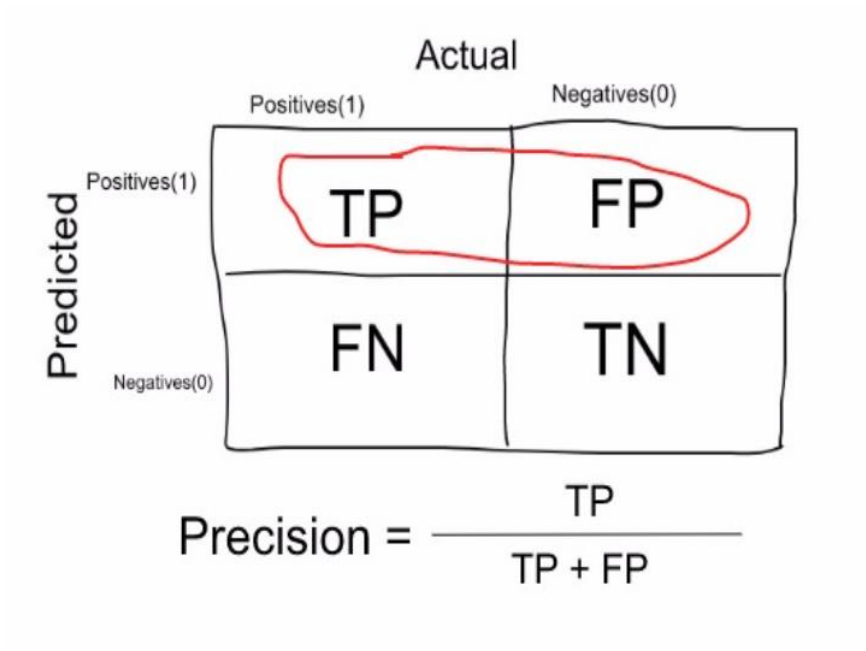


Figure 7.1 Confusion Matrix



$$\text{Accuracy} = \frac{TP + TN}{TP + FP + FN + TN}$$

Figure 7.2 Accuracy



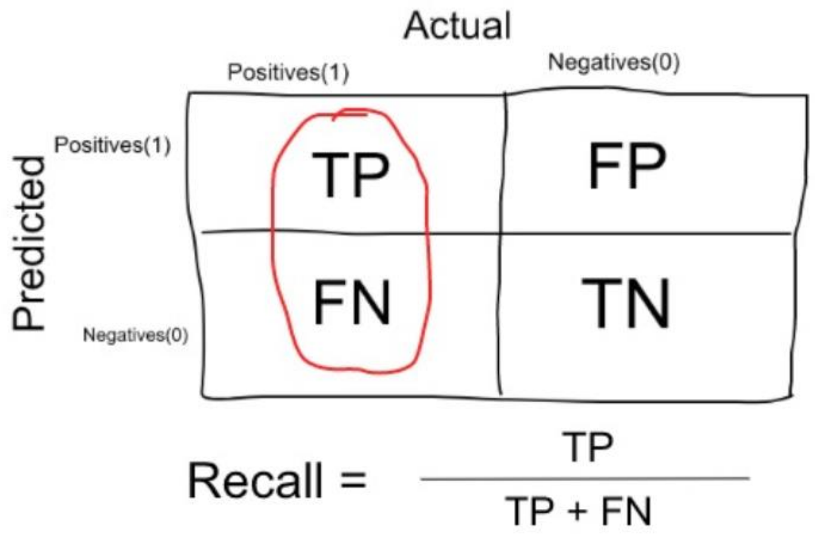


Figure 7.3 Precision and Recall

The harmonic mean of precision and recall is the F1 Score. It informs you of the classifier's precision, or how many times it accurately categorizes, as well as its robustness. High precision but poor recall gives you an incredibly precise result, but it still misses a vast amount of incidents. The higher the F1 Score, the stronger our model's results, because it practically attempts to bridge the gap among recall and precision. F1_score from sklearn.metrics is the module used in our implementation.

One of the most commonly used metrics for measurement is the Area Under Curve (AUC). It's used to solve problems involving binary classification. A classifier's AUC is the likelihood that a selected randomly positive example will be ranked higher than a randomly picked negative example. A receiver operating characteristic curve, or simply ROC curve, is a visual representation of the output of a binary classifier scheme as the threshold is changed. It's made by comparing the fraction of true positives to the fraction of false positives at different threshold settings. In other words, the AUC-ROC metric measures a model's ability to differentiate between classes, which means the stronger the model, the higher the AUC.

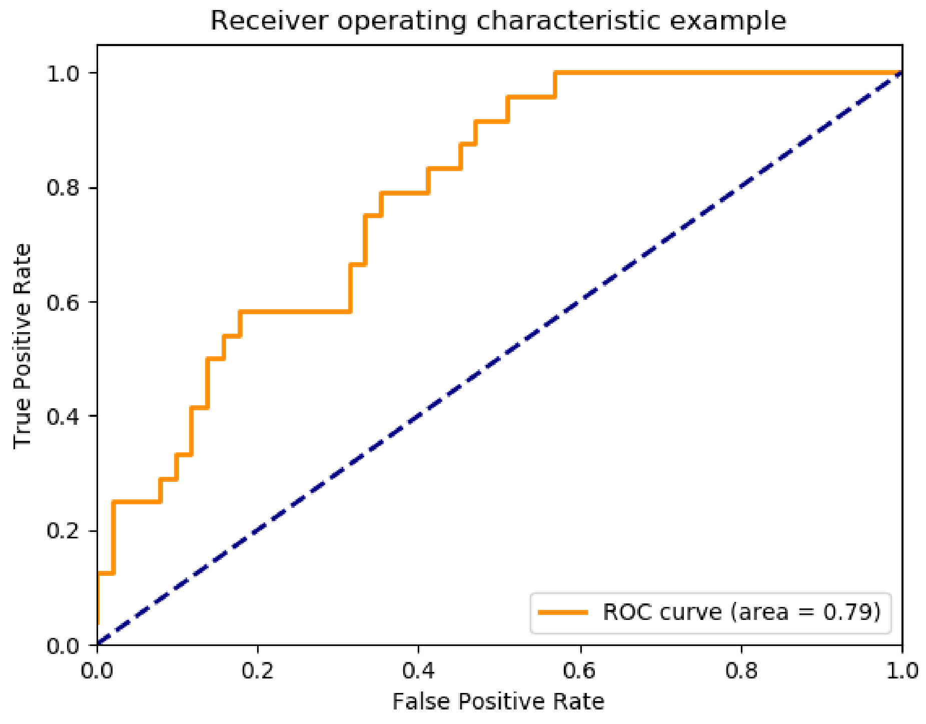


Figure 7.4 ROC Curve

6.2. Evaluation

Below are presented in a detailed manner the results of multiple different tests, as explained earlier, on Omnidroid dataset. To sum up, the experiments that were eventually run and are shown in these tables, are the following (the dataset and the features used are the same on all runs and are shown in 4.1.1.):

1. Random Forest, Extra Trees, Gradient Tree Boosting, ADA Boost and proposed stacked classifier exclusively and separately each one of them on static features
2. Random Forest, Extra Trees, Gradient Tree Boosting, ADA Boost and proposed stacked classifier exclusively and separately each one of them on dynamic features
3. Fusion of both static and dynamic feature results using soft-voting as final classifier and the proposed stacked classifier as the classifier that was used on both static and dynamic features (since it had the best results, seen also in Figure 7.5 and 7.6). The W_S and W_D abbreviations following figure 7.7, represent the weight of static and the weight of dynamic features respectively, all tested in a range between and 0.1 and 0.9.

On the first table, it can be seen that the stacked classifier outperforms all the other ensemble classifiers in a class of at least 2%. Same thing goes with the dynamic features, as it can be spotted on figure 7.6 below. Finally, using stacked classifiers on both the feature categories, the voting classifier fuses the results giving even better results, with maximum performance obtained at W_S = 0.7 and W_D = 0.3, going as far as 5% better results.

	Accuracy	**Precision**	**Recall**	**F1**	**AUC**
Random Forest	0.875700	0.878454	0.867739	0.872952	0.951470
Extra Trees	0.876200	0.881434	0.864897	0.872999	0.948090
Gradient Tree Boosting	0.875400	0.882224	0.862051	0.871948	0.946220
ADA Boost	0.849700	0.8444737	0.851078	0.847870	0.927380
Stacked Classifier	0.890800	0.893609	0.871032	0.886649	0.965431

Figure 7.5 Static Features Evaluation

	Accuracy	**Precision**	**Recall**	**F1**	**AUC**
Random Forest	0.765500	0.760719	0.742282	0.750964	0.834027
Extra Trees	0.753167	0.760315	0.745570	0.747207	0.826118
Gradient Tree Boosting	0.758000	0.751021	0.750000	0.739962	0.831020
ADA Boost	0.736900	0.735019	0.735019	0.726349	0.811444
Stacked Classifier	0.778833	0.774882	0.745302	0.754182	0.852121

Figure 7.6 Dynamic Features Evaluation

	W_S = 0.1 W_D = 0.9	W_S = 0.2 W_D = 0.8	W_S = 0.3 W_D = 0.7	W_S = 0.4 W_D = 0.6	W_S = 0.5 W_D = 0.5	W_S = 0.6 W_D = 0.4	W_S = 0.7 W_D = 0.3	W_S = 0.8 W_D = 0.2	W_S = 0.9 W_D = 0.1
Accu-racy	0.845800	0.865900	0.865800	0.885732	0.907739	0.908730	**0.915200**	0.907730	0.905600
Preci-sion	0.854664	0.863000	0.864464	0.883426	0.894890	0.895100	**0.902210**	0.902440	0.892300
Recall	0.852550	0.861050	0.861430	0.877710	0.892050	0.872324	**0.899176**	0.898700	0.895145
F1	0.842350	0.863350	0.863400	0.878800	0.894500	0.904450	**0.913405**	0.904740	0.903630
AUC	0.957030	0.958000	0.958550	0.963548	0.965035	0.971034	**0.975730**	0.973030	0.966030

Figure 7.7 Voting Fusion Evaluation (using stacked ensemble as sub-classifiers)

7. Future Work

Several suggestions for future work are discussed in this segment, which can enhance the identification of Android malware. Having analyzed this dataset, we conducted various experiments, and presented findings after analyzing groups of features independently in order to supply relevant implications with a preliminary assessment of the dataset, as well as illustrating the dataset's large capacity and efficiency of use. About the fact that most of the state-of-the-art algorithms studied, performed well in the experiments, there is also some space for progress as compared to the OmniDroid dataset [18]. This could aid researchers in training, testing, and evaluating their models, or simply comparing malware detection approach using OmniDroid as a safe, pre-processed data standard. Ultimately, as mentioned in [18], there is an idea about expanding AndroPyTool's structure to allow for the extraction of more features per sample by adding other feature extraction and reverse engineering methods. Simultaneously, there is a need in the immediate future to increase the number of samples in the OmniDroid dataset, upgrading it to include new samples discovered in the wild.

In addition to that, using stacked classification models and having more than one algorithm running on the same data, consumes more time, as already mentioned. A solution to this issue is apparently running those algorithms in parallel which of courses provides a great aid but requires having the correct amount of computation power. So, with regard to how much time is required and the performance implications, this kind of models need direct improvement.

In the long run, we want to build further concurrent classifiers by combining new variations of individual classifiers. Our aim is to use deep-learning models on a massive real-world dataset to identify and forecast security flaws and malware. We want to use the examined technique to build and test a complete Android malware detection mechanism in the future. More research into the detection engine's success when exposed to real datasets from evolving Android app markets is also expected.

8. Conclusion

Android mobile applications are vulnerable to malware attacks, and while studies have extensively dealt with malware detection and classification, their methods could be improved [3]. As a result, this thesis tries first of all, to encourage learning about Android subsystems and its security architecture, while ensuring the criticality regarding malware and the procedure of correctly analyzing and engineering. On a second level, it intends to show how hybrid parallel ensemble classifiers like a stacked model and soft voting, can be used to efficiently detect and identify malware in Android applications, while promoting the correct use of datasets for testing purposes, wanting to encourage the research on Android malware analysis even more. In practice, individual classifiers are used throughout our approach, which ensembles them to construct a more efficient and reliable system based on a stacked model, leading to a linking process between dynamic and static results.

First, a theoretical context was given, including theoretical knowledge and related work on machine learning classification techniques, in order to better develop the machine learning classifiers. After that, the solution to the problem was presented. Finally, the implementation algorithm was introduced, as well as the empirical results of the output metrics. Although there have already been numerous researches for identifying Android malware using machine learning techniques, as discussed before, the aim of this work was to show how stacked methods combined with parallel voting between dynamic and static features can improve the results in general.

Finally, the system that was tested, not only detects malware in the Android OS with a very high accuracy and precision, but it also eliminates the gaps and shortcomings of previous methods. This approach takes advantage of each classifier's unique strengths and strong points when averaging out their individual flaws. As a result, the system as a whole is more stable and has a lower error rate. Our study, in general, involves data for developing machine learning models and recommendations given for machine learning classification of Android malware that has been shown to increase classification accuracy.

References

[1] Aafer, Y., Du, W., Yin, H.. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In: 9th International ICST Conference, SecureComm 2013; vol. 127 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer; 2013, p. 86–103.

[2] Agrawal, Prerna & Trivedi, Bhushan. (2021). Machine Learning Classifiers for Android Malware Detection. 10.1007/978-981-15-5616-6_22.

[3] Ahmed, Omar & Sallow, Amira. (2019). Android Security: A Review. 6. 6.

[4] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., et al. Flowdroid: precise context, flow, field, objectsensitive and lifecycle-aware taint analysis for android apps. In: O'Boyle, M.F.P., Pingali, K., editors. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14. ACM; 2014, p. 259–269.

[5] Ayodele, Taiwo. (2010). Types of Machine Learning Algorithms. 10.5772/9385.

[6] Droidbox source code repository. 2011. URL: https://github. com/pjlantz/droidbox; Last accessed: 2018-03-14.

[7] Drucker. "Improving Regressors using Boosting Techniques", 1997.

[8] Dunham, K. & Hartman, S. & Morales, J.A. & Quintans, M. & Strazzere, T.. (2014). Android malware and analysis. 10.1201/b17598.

[9] Ethem Alpaydin (2020). Introduction to Machine Learning (Fourth ed.). MIT. pp. xix, 1–3, 13–18. ISBN.

[10] Faruki P, Ganmoor V, Laxmi V, Gaur MS, Bharmal A. AndroSimilar: robust statistical feature signature for Android malware detection. In: Proceedings of the 6th international conference on security of information and networks. ACM; 2013. p. 152–9. doi: 10.1145/2523514.2523539.

[11] https://github.com/pandas-dev/pandas

[12] Google n.d. Android Developer Guide. http://developer.android.com/guide/index.html. Accessed September 25, 2020.

[13] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, “Crowdroid: behaviorbased malware detection system for android,” in Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices. ACM, 2011, pp. 15–26.

[14] Jonathan Levin, Android Internals: A Confectioner's Cookbook, Volume I: The Power User's View, http://newandroidbook.com/AIvI-M-RL1.pdf, 2015.

[15] Kapratwar, Ankita & Di Troia, Fabio & Stamp, Mark. (2017). Static and Dynamic Analysis of Android Malware. 653-662. 10.5220/0006256706530662.

[16] Liu, Jianye & Yu, Jiankun. (2011). Research on Development of Android Applications. Proceedings - 2011 4th International Conference on Intelligent Networks and Intelligent Systems, ICINIS 2011. 10.1109/ICINIS.2011.40.

[17] L. Breiman, “Bagging predictors”, Machine Learning, 24(2), 123-140, 1996.

[18] Martín García, Alejandro & Lara-Cabrera, Raul & Camacho, David. (2018). Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset. Information Fusion. 52. 10.1016/j.inffus.2018.12.006.

[19] Mitchell, Tom (1997). Machine Learning. New York: McGraw Hill. ISBN 0-07-042807-7.

[20] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer, “Andrubis–1,000,000 apps later: A view on current android malware behaviors,” in Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), 2014 Third International Workshop on. IEEE, 2014, pp. 3–17

[21] Opitz, D.; Maclin, R. (1999). "Popular ensemble methods: An empirical study". Journal of Artificial Intelligence Research. 11: 169–198.

[22] O. Matan: On voting ensembles of classifiers, In: Proc. of the 13th Natl. Conference on Artificial Intelligence, 84–88, 1996.

[23] P. Feng, J. Ma, C. Sun, X. Xu and Y. Ma, "A Novel Dynamic Android Malware Detection System With Ensemble Learning," in IEEE Access, vol. 6, pp. 30996-31011, 2018, doi: 10.1109/ACCESS.2018.2844349.

[24] Rasthofer, S., Arzt, S., Bodden, E.. A machine-learning approach for classifying and categorizing android sources and sinks. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014. The Internet Society; 2014, p. 1–15.

[25] Scikit-learn python library, https://scikit-learn.org/stable/user_guide.html

[26] Sill, J.; Takacs, G.; Mackey, L.; Lin, D. (2009). "Feature-Weighted Linear Stacking".

[27] Simeone, O., A Brief Introduction to Machine Learning for Engineers, 2017.

[28] S. Arshad, M. A. Shah, A. Wahid, A. Mehmood, H. Song and H. Yu, "SAMADroid: A Novel 3-Level Hybrid Malware Detection Model for Android Operating System," in IEEE Access, vol. 6, pp. 4321-4339, 2018, doi: 10.1109/ACCESS.2018.2792941.

[29] Tam, Kimberly & Feizollah, Ali & Anuar, Nor & Salleh, Rosli & Cavallaro, Lorenzo. (2017). The Evolution of Android Malware and Android Analysis Techniques. ACM Computing Surveys. 49. 1-41. 10.1145/3017427.

[30] Van der Veen, Victor. (2013). Dynamic Analysis of Android Malware. 10.13140/2.1.2373.4080.

[31] Wolpert, David H. "Stacked generalization." Neural networks 5.2 (1992): 241-259.

[32] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," ACM Transactions on Computer Systems (TOCS), vol. 32, no. 2, p. 5, 2014.

[33] Yan L-K, Yin H. DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android Malware analysis. In: USENIX security symposium; 2012. p. 569–84.

[34] Zhauniarovich, Yury. (2014). Android Security (and Not) Internals.