Android malware detection based on system calls: the current state

April 27, 2018

Department of Mathematics and Computer Science, Faculty of Sciences / University of Ngaoundéré

Authors:

- Kodeh Forey Clement
- Ngoran Magnuss Dufe
- Faissal Ahmadou
- Wibada Suzanne
- Meidogo Seinbero Christophe
- Djakene Wandala Albert
- Feuyo Manfouo Raissa Edwige
- Sokdou Bila Lamouyannick
- Vandi Koyang Joseph
- Gongmonga Dissala
- Djebe Narbe Yves
- Kaweina Sanigue Joas
- Ntsama Jean Emmanuel

Mentor: Dr._Ing. Tchakounte

Abstract

The Android platform is a favourable choice for most mobile device manufacturers due to its open source nature. Making the Android operating system so popular in use in the world that is increasingly "mobile", with the use of Smartphones, wearable devices, and portable tablets being the order of the day. Billions of applications developed for this platform in various domains of life attract a lot of malicious agents. There is therefore a great need to guarantee the security of the resources, activities and privacy of the users of the Android operating system. A lot of work is being done to this effect, ranging from static to dynamic approaches to detect Android malware. In this paper, we present the current state of dynamic malware detection using system calls. We begin with a background on android ecosystem, processes and threads, and system calls. Then we look at feature engineering and detection ecosystem. We then study existing work that has been done over the years in this area, and use various elements of the detection ecosystem such as sources of datasets, environments for detection, criteria and measures of performance and detection approaches to compare these methods of malware detection via System Calls. We notice that most of the methods use a limited number of benign and

malware applications in their test experiment to obtain high values of performance. However, the detection methods are far from being real-time, and often require third party environments like servers and the cloud, with no guarantee of the users' resource privacy.

Keywords: malicious agents, Android malware; dynamic malware detection; system calls; security threats; classification algorithm; evaluation criteria.

Introduction

The open nature of the android operating system has favoured its choice as platform for many different mobile device manufacturers. This has led to a great popularity in use in a world that is increasingly "mobile"; with the use of Smartphones, wearable devices, and portable tablets being the order of the day. This platform supports various apps developed by both mobile device manufacturers and third party developers. However, these people cannot assure the total security of the apps they develop, exposing users to a lot of security threats such as system resource occupation, user behaviour surveillance, and user privacy intrusion. Also the popularity of this platform attracts a great number of malicious agents. The malware on the android platform must be detected to secure billions of users of mobile devices. Static methods aimed at finding malicious characteristics or suspicious code segments in android applications have given way to dynamic analysis that focuses on the behaviour of the application at runtime. The characteristics of mobile devices like limited resources and constant connectivity, opens so many doors for mobile malware, creating new detection challenges, especially with dynamic runtime malware detection. Various approaches exist for android malware detection that focus on hardware use (Battery, CPU, memory, I/O...), software(permissions, privileges, network traffic ...), firmware(system calls, IPL,API library ...) and many others. System calls are one of the most used features in android malware detection. The patterns that a malware present during a system call and the predefined functions manipulated can be explored to detect malware.

In this work, we shall focus on the current state of android malware detection by system calls. We begin by presenting a background on android ecosystem, processes and threads and system calls. In the second part we shall focus on feature engineering: feature extraction, event generation and machine learning. In the third part we shall look at the detection ecosystem. The fourth part compares current detection methods based on system calls and we end our work by presenting open issues and future research trends.

Related work

Some work has been done to survey dynamic malware detection. Zheng Yan et al [13] presented a thorough survey on dynamic mobile malware detection. Where they introduced the definition, evolution, classification, and security threats of mobile malware, summarize a number of criteria and performance evaluation measures of mobile malware detection and compared, analyzed and comment existing mobile malware detection methods proposed in recent years based on the evaluation criteria and measures, and figured out open issues in this research field and motivate future research directions. However, their works based on dynamic malware detection in general. We focus on Android malware detection using system calls which is more specific.

Part I
Background

1 Android ecosystem

The android is an open source platform for mobile, embedded and wearable devices maintained principally by Google. The android ecosystem is made up of the following components:

- The mobile network operators that install, maintain and operate mobile networks, defining communication protocols and providing network services;
- The networks supported by the android mobile devices such as 2G, GSM, 3G, 4G, LTE and WiMAX;
- Android devices, with mobile device features like telephone, camera, Bluetooth, GPS, Accelerometer and storage;
- The android platform that runs Dalvik Virtual machine based apps;
- The Android Operating system built over the LINUX kernel that integrates various drivers;
- The Android Application framework that can be explored by all applications that can run on the Dalvik virtual machine.
- Android Applications which are programs that users install on the device and interact with (Apps) like browsers, contact managers and call managers;
- Services such as call services, SMS services, internet access and location tracking and
- User, who are the people that operate the android devices.

1.1 The android architecture

The android architecture is made up of the following layers:

- Kernel- It contains all the low-level device drivers for interacting with the underlying Hardware.
- Libraries - Contains all the code that provides the main features of android OS. SQLLite library provides database support so that an application can store data. Webkit is to be used for web-browsing.
- AndroidRuntime- Provides a set of core libraries that enable developers to write Android Apps using Java programming language. A specialized virtual machines named as Dalvik is a core component.
- Application framework- this exposed the different API's so that App developers can make use of them in the application.
- Applications - A layer to depict all the last mile value added applications.

As shown in the figure below

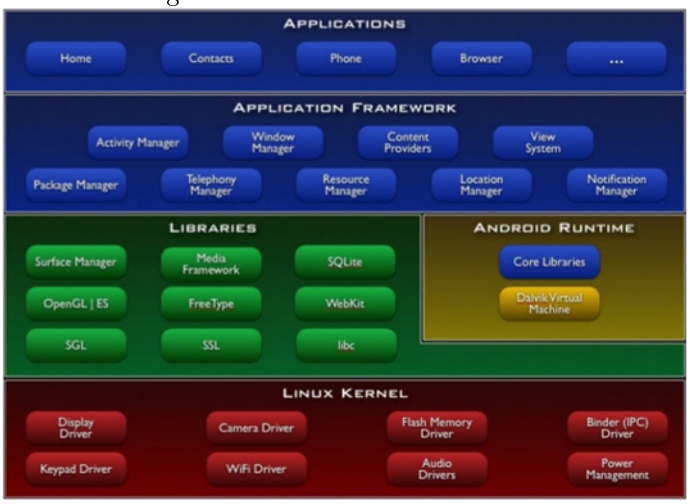


Figure 1: android architecture

1.2 The Android application structure

It consists of :

- Activity manager that controls visible components of applications;
- Content provider that encapsulates data that need to be shared between applications (via custom interfaces);
- Resource manager responsible for all the resources necessary for the application to run;
- Location manager in charge of locations;
- Notification manager that follows events such as arriving messages, appointments, proximity alerts.

1.3 Pre-defined system security schemes in Android

Security schemes are put in place with major Android releases and monthly security updates to ensure the protection of applications, user data and system resources as well as the operating system isolation from user programs and users. These schemes operate at the system level and user level. These security schemes include:

- Operating system based security (kernel self protection) through the Linux kernel that is stable and secure thanks to its widespread use and security research.
- Mandatory application sandbox for all applications to control interactions between applications, ensuring security at the process level based on UNIX-style user separation of processes and file permissions. This keeps each application in a separate space protecting its data and operations from other applications.
- Secure inter-process communication that prevents applications from obtaining unauthorized data from other applications overhead.
- Permission based security where Android devices are protected by a permission based framework that restricts some applications that attempts to access sensitive resources like contacts, SMSs and some storage locations in the devices. This permission is generally granted at install time
- There is a system partition that contains the Kernel, libraries, applications runtime, framework and applications set to read only. When the device is in safe mode third party applications are not launched by default assuring some degree of security.
- Security Enhanced Linux (SELinix) to ensure mandatory access control on processes above the kernel.
- Hardware backed security that strengthens the main storage and cryptographic systems and enable strong remote authentication.
- The presence of verified boot and device-mapper-verity in Android 6.0 and later versions guarantees system software integrity by verifying each boot stage cryptographically for integrity authenticity during boot. This ensures that the operating system starts in a known good state.
- A set of cryptographic APIs are provided for use by applications, including standard cryptosystems like RSA, SHA, DSA, and AES. This protects data from unauthorised users.
- File system Encryption to encrypt all data. The presence of full-disk encryption protects the entire device user data.
- The presence of an Android Device Administration API schemes enables administrators to enforce password policies across devices, providing security.
- User space hardening that protects the operating system and applications against memory corruption vulnerabilities through mechanisms like Address Space Layout Randomisation (ASLR) and Data Execution Prevention (DEP).

- Secure lock screen that uses patterns, finger prints and password to access the device has reinforced security

2 Processes and Threads

2.1 Processes

A process is an instance of a program in execution. All components of the same application run in the same process. The android system tries to maintain an application process for as long as possible but eventually needs to remove old processes to reclaim memory for new or more important processes. To determine which processes to keep and which to kill, the system places each process into an importance hierarchy based on the components running in the process and the state of those components. Processes with the lowest importance are eliminated first, then those with the next lowest importance, and so on, as necessary to recover system resources. The following list presents the different types of processes in order of importance:

- **Foreground process**, required for what the user is currently doing.
- **Visible process** that can affect what the user sees on screen.
- **Service process**, that runs a service that has been started and does not fall into either of the two higher categories. For example processed that enable us to play music in the background or download data on the network
- **Background process** A process holding an activity that's not currently visible to the user . Usually there are many background processes running, so they are kept in an LRU (least recently used) list to ensure that the process with the activity that was most recently seen by the user is the last to be killed.
- **Empty process** that doesn't hold any active application components. It is kept for caching purposes, to improve startup time the next time a component needs to run in it.

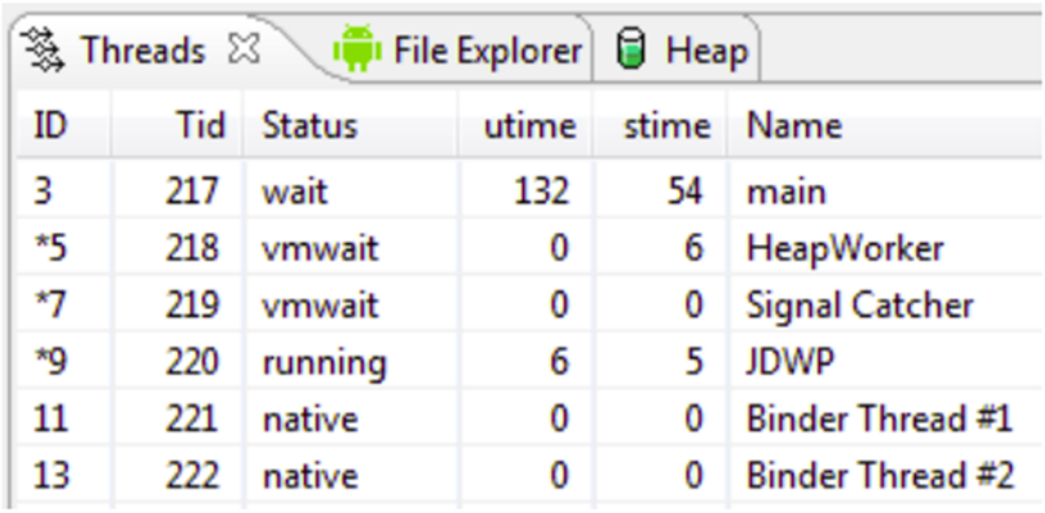
2.2 Threads

A thread is a portion of code that can be executed independently of the main program. The threads of a process share its executable code. When an Android application is launched, the system creates a thread of execution for the application, called "main" or UI thread. All components that run in the same process are instantiated in the UI thread, and system calls to each component are dispatched from that thread. Consequently, methods that respond to system always run in the UI thread of the process.

States of a thread

When a thread is created, it is in the new state. It goes to executable state when it is in the election queue while waiting to be (possibly again) chosen by the scheduler. It gets to the running state when it has been chosen by the scheduler in its time quotas. When this quota is reached, it returns to executable and returns to the election queue. It is in a blocked state if it is waiting for a resource like input/output. It dies when he finishes executing its last instruction. A thread may have a priority level in order to be chosen more or less quickly by the scheduler.

Davy Leggieri (2010) [23] The following image shows us the characteristics of the processes on Android.



ID	Tid	Status	utime	stime	Name
3	217	wait	132	54	main
*5	218	vmwait	0	6	HeapWorker
*7	219	vmwait	0	0	Signal Catcher
*9	220	running	6	5	JDWP
11	221	native	0	0	Binder Thread #1
13	222	native	0	0	Binder Thread #2

Figure2 :The characteristics of the processes on android.

3 System Calls

A system call is the mechanism used by applications to request for a service from the operating system kernel. The Kernel is the core of the operating system. Kernel provides a simple means for the application programs to interact with the hardware through system calls. System calls provide an interface between processes and operating system. The operating system provides services, including the creation and execution of new processes and access control of resources. The sequence of system calls occurs consecutively over time and can capture actions performed by applications during execution.

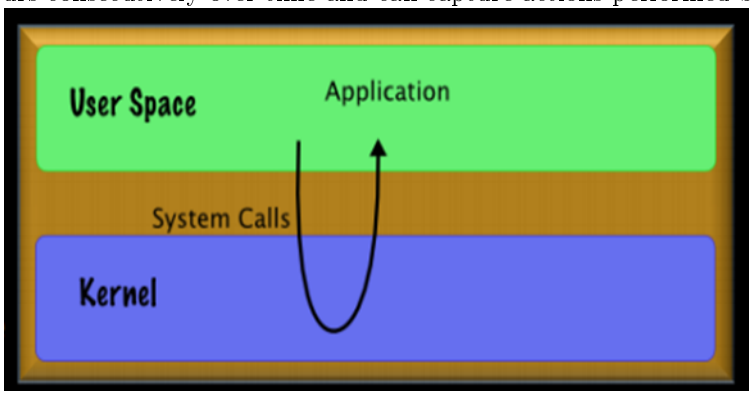


Figure3: system call

Whenever a program invokes a system call, it is interrupted and its context is saved. The Kernel carefully checks that the request is valid and that the process invoking the system call has enough privilege. If all is well, the CPU then starts executing in the kernel mode and a routine corresponding to the system call is executed. After the completion of that routine the control is transferred to the user mode and the program is restored. Some system call operations are exposed to the end user in the form of simple library calls/APIs such as - read(), write(), open() in libc. The main of such operations is executed by the OS kernel code after switching the mode (TRAP instruction) from user to kernel. These and several other system call operations typically access/manipulate OS kernel data structures like buffer cache, inodes, process control block, global open file table and many others. System calls therefore make communication between operating system and hardware, processes and file access possible.

3.1 The main purposes of system calls

The main purpose of system calls is to enable applications to be able to communicate with the operating system, prompting the operating system to carry out a service. For example, if an application wants access to a hardware resource, it "calls" the operating system to read/write from/to an I/O or storage device.

3.2 Types of systems calls

System calls can be grouped into 5 categories: process control, file manipulation, device manipulation, information maintenance and communication.

- Process control calls that handles running programs.
- File management system calls to carryout file manipulation operations such as creating, deleting, reading, writing, repositioning, or closing a file.
- Input/Output(device) management that allocates device resources to the user programs when they request it and takes them back when they are through.
- Information management for transferring information between the user program and the operating system.
- Communication system calls forinter-process communication. This can be done via the message-passing model and the shared memory model.

3.3 Android Kernel and System Calls

The android applications take the kernel services through system calls. Whenever a user requests for a service like "call a phone" in user mode through the phone call application, the request is forwarded to the Telephone Manager Service in the application framework. The Dalvik Virtual Machine in Android runtime transforms the user request passed by the Telephone Manager Service to library calls, which results in multiple system calls to Android Kernel. While executing the system call, there is a switch from user mode to kernel mode to perform the sensitive operations. When the execution of operations requested by the system call is completed, the control is returned to the user mode.

Part II
Feature engineering

4 Feature Extraction

Feature extraction involves transforming large amount of input into a set of features. Feature extraction is useful in a situation where there is much input data into an algorithm, especially redundant and irrelevant data. It gives precise measurement of features which helps in the classification of input as malicious or benign. It transforms the features into an organised and more manageable subset of information reducing the dataset for processing.

Some features that can be extracted include:

- Byte n-gram Features: these are sequences of n bytes extracted from malware used as signature for recognising malware. It yield high accuracy in detecting new malware.
- Opcode n-gram feature: this feature reveal statistical diversities between malicious and legitimate software and can be is a more efficiently and effectively classified.
- Portable executables: extracted from certain parts of executable files. They are extracted by static analysis using structural information of feature extraction.
- String features: these are plaintext based features encoded in executables like windows, getversion, lkbrary etc.

- Function based features: these are features extracted over the runtime behaviour of the program files. They explore features that reside in a file for execution and utilise them to produce various attributes representing the file. These features include system calls, Windows Application Programming Interface(API) calls , their parameter information flow tracking and instruction set. etc.

Different methods can be used to extract various features from different system calls. Malware and trusted applications have different behaviours as regards system calls. For example, a malicious Android application may request more permissions or access sensitive resources more frequently. The system calls made by these applications are not independent, they may involve several other system calls sequentially. The system calls an application makes and the dependencies amongst the system calls can be captured and explored to obtain the dynamic behaviors of applications. The different system call sequences reflect different behaviors. As such, system call sequences can be used to extract features for malware detection.

Methods used for Feature Extraction

4.1 Frequency vector:

In this method, a feature vector is used to represent features. Every feature in the vector represents the number of occurrences of a system call during the execution of an application.

Given a system call sequence, a feature vector x = [x1, x2, ... , xj, Sj], can be defined, where xi is equal to the frequency of system call si in s

System call frequency models only consider the independent system calls occurrence times and ignore the relationship between system calls.

4.2 The co-occurrence matrix:

In this method, a co-occurrence matrix of a sequence of system calls is built.

For example let M be the co-occurrence matrix of a sequence (X=x1, x2,x3,...,xN) with length N. And let (s1, s2, s3, ..., sn) represent N system calls . The correlation between si and sj can be computed by counting the occurrence times of the system call pair within a scope size of k. A co-occurence matrix can be constructed by doing this for each system call pair. This matrix can then be normalised. However, the normalized co-occurrence matrix cannot be fed into a classifier model directly as the input of most classifiers is a vector. It needs to be transformed into a vector.

Results from experiments [18] have shown that detection results of the co-occurrence matrix are much better than those of the frequency vector.

5 Event Generation

5.1 Event

Events are a useful way of collecting data on a user's interactions with an application's component(User Interfce). These interactions include the pressures on the buttons or the support on the screen, etc. the Android infrastructure keeps a queue of events as first in first out (FIFO). These events can be captured into an application and appropriate actions taken. There are three concepts related to Android event management

1. **Event Listeners:** An event listener is an interface in the View class that contains a single method recall. These methods will be called by the Android infrastructure when the view to whichthe listener was recorded is triggered by the user's interaction with the item inthe user interface.

2. **Event listener registration:** Event recording is the process by which an event handler is registered with an event listener so that the handler is called when the event listener triggers the event.

3. **Event Managers:** When an event occurs and we have recorded an event listener for the event, the event listener calls the event handler, which is the methodwho actually manages the event.

Applications consist of four basic types of elements related to both internal and external events:

1. **Activities :** Activities react to user input events such as clicks, and therefore are the main target of the test tools for Android.

2. **The services** : Service run in the background, separate from the main thread of the UI and are useful for operations that should not affect the threaduser interface, for example, downloading content. Unlike activities,they do not provide a user interface, and therefore, they are usually nota direct target of Android test tools, but they could be tested indirectly by certain activities.

3. **Broadcast receivers** are used to listen to system-wide events, such as incoming SMS, phone calls or emails.

4. **Content providers** allow applications to make data availablefor use by other applications. The behavior of an application can depend on the status of the content providers (for example, if a contact list is empty or if it contains duplicates). As a result, the test tools could search the content providers 'mockery' in an attempt to do deterministic testing and get better coverage of the behavior of an application.

Android apps are event-driven, i.e, the program is run in the idle state waiting for the user to interact with the application via the graphical interface,or some type of system events from the Android OS. In Android, events GUIs include actions by the user such as clicks, rolls, orsystem events, such as updating GPS location.

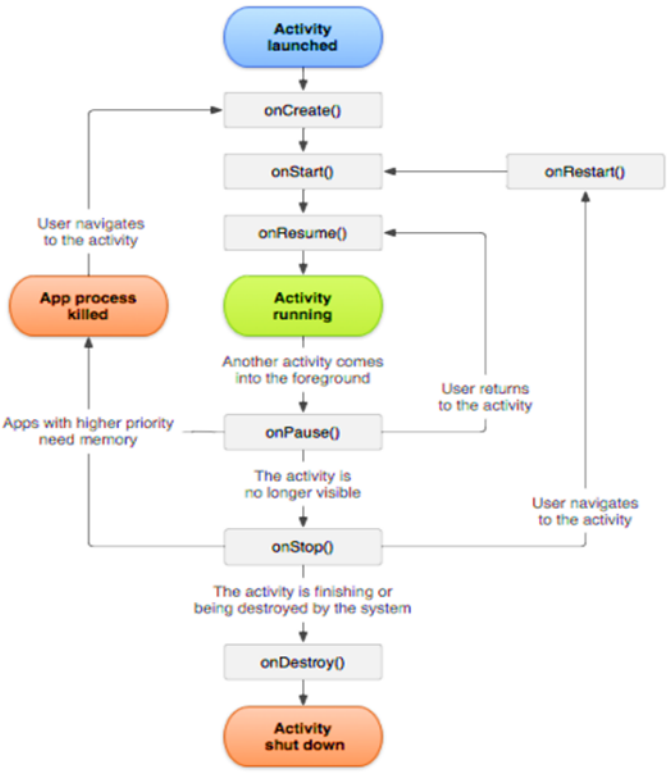


Figure4: presentation of the cycle of life of an activity

5.2 Event Generation

The main goal of input generation tools for Android is to detect the existing faults in the applications being tested. Application developers are usually the main actors of these tools to automatically test their applications and discover problems before deploy. The dynamic traces generated by these tools, can be the starting point for event generation. Events can be generated using analytic and static tools such as Monkey, Dynodroid, DroidFuzzer etc.

5.2.1 Monkey

Monkey is the most frequently used tool for testing Android apps, partly because it is part of the Android developer's toolkit and does not require additional installation effort. Monkey implements

the most random strategy because it considers the application to be tested as a black box and can only generate UI events

5.2.2 Dynodroid

Dynodroid is a system that automatically generate relevant entries for Android apps. It is able to generate both the user interface entries(for example, touch screen keys and controls) and system inputs (for example, the simulation of messages, incoming SMS) and it does so by checking those which are relevant for the application. The main underlying principle used by Dynodroid is: observe, select and execute cycle in which it first observes which events are relevant for the application in the current state, then selects one of these events, and finally runs the event selected to give a new state in which it repeats this process.

The table[24] gives an overview of the existing test entry generation tools for Android presented in the table below.

Name	Available	Instrumentation		Events		Exploration strategy	Needs source code	Testing strategy
		Platfom	App	UI	System			
Monkey	✓	o	o	✓	o	Random	o	Black-box
Dynodroid	✓	✓	o	✓	✓	Random	o	Black-box
DroidFuzzer	✓	o	o	o	o	Random	o	Black-box
IntentFuzzer	✓	o	o	o	o	Random	o	White-box
Null IntentFuzzer	✓			✓	✓	Random		Black-box
GUIRipper	✓	o	✓	✓	o	Model-based	o	Black-box
ORBIT	o	?	?	✓	✓	Model-based	✓	Grey-box
AE-Depth-First	✓	✓	✓	✓	o	Model-based	o	Black-box
SwiftHand	✓	✓	✓	✓	o	Model-based	o	Black-box
PUMA	✓	✓	✓	✓	o	Model-based	o	Black-box
AE-Targeted	o	o	✓	✓	o	Systematic	o	Grey-box
EvoDroid	o	o	✓	✓	o	Systematic	o	White-box
ACTEve	✓					Systematic		White-box
JPF-Android	✓	o	o	✓	✓	Systematic	✓	White-box

Table overview of the existing test entry generation tools for Android

5.3 Machine Learning

Machine Learning is a domain of computer science that seeks to give computers the ability to learn from data instead of being explicitly programmed. It can be split into two major methods supervised learning and unsupervised learning . Supervised learning works on labeled data while unsupervised learning works on unlabelled data. Classification algorithms make use of supervised learning to classify data. Classification it can either be binary (malware-not malware) or multi-class (cat-dog-pig-lama. . .) thus malware detection falls under binary classification.

5.3.1 Machine Learning Algorithm

Regarding the algorithms implemented for the detection of malicious and non-malicious system calls, several algorithms have been implemented on the basis of the SVM (Support Vectors Machine or Well-Separators at Large Margins) architecture. This means that all types of system calls are listed in two sets, those that are potentially malicious and those that are not. For our case study, we chose to present the so-called hybrid algorithm approach of Fei Tong and Zheng Yan[6] , which is a two-level algorithm.

Algorithme 1 : Generation of malicious and pattern sets

$MP = NP = \Phi$;

For $\forall$ k $\in$ {sequential system calls with different depth}

If $MW_k^n = {MF_k^n}/{NF_k^n} > tm$, put k into MP, else if $NW_k^n = {NF_k^n}/{MF_k^n} > tn$, put k into NP.

Note: In case that MF_k^n or NF_k^n only appears in MS or NS, i.e., MW_k^n or NW_k^n is infinite ∞, we put k into MP or NP.

Figure5: Presentation of algorithm

Description of the algorithm

As the name suggests, this algorithm generates a set of malware models.

Input : It takes as input:

- A value in a sequence of k system calls in a MS (Malware Set) set;
- another value in a sequence of system calls k in a set NS (Benign app Set);
- A variable, tm, which is the threshold for judging a model of malware;
- A variable, tn, which is the threshold for judging a normal software model.

With k belonging to a sequence of system calls with different depths, with n an integer.

Output: It returns a set of Malware pattern (MP) , and a set of normal pattern(NP) software.

Algorithme 2 : Malware detection

$Qm = Qn = 0$.

For $\forall$ uk $\in$ {sequential system calls of a },

If uk = k, k $\in$ MP, $Qm = Qm + MW_k^n$;

Else if uk = k, k $\in$ NP, $Qn = Qn + NW_k^n$.

If $Qm > Mt$ and $Qn < Nt$, the app is malware.

If $Qm < Mt$ and $Qn > Nt$, the app is normal.

If $Qm > Mt$ and $Qn > Nt$, cannot judge.

If $Qm < Mt$ and $Qn < Nt$, cannot judge.

Figure 6: Presentation of the algorithm

Description of the algorithm

Input: It takes as input:

- An unknown application for detection;
- Nt represents the normal correspondence threshold;
- Mt represents the threshold of malicious correspondence;
- MP which is the set of malware pattern;
- NP which is the normal software pattern set;

With uk belonging to a sequence of system calls with different depths, with n an integer.

Output: Detection result

Note

The hybrid aspect is clearly seen because algorithm 2 uses the Malware MP model set, and the normal NP software model set returned by algorithm 1 as the basis for learning to test on. any software introduced into the detection system.

Evaluation metrics

- We seek to predict Y (with value in {-1 / +1}) in terms of the variables X1, ... Xp.
- We look for the model such that P ((X) # Y) minimum.

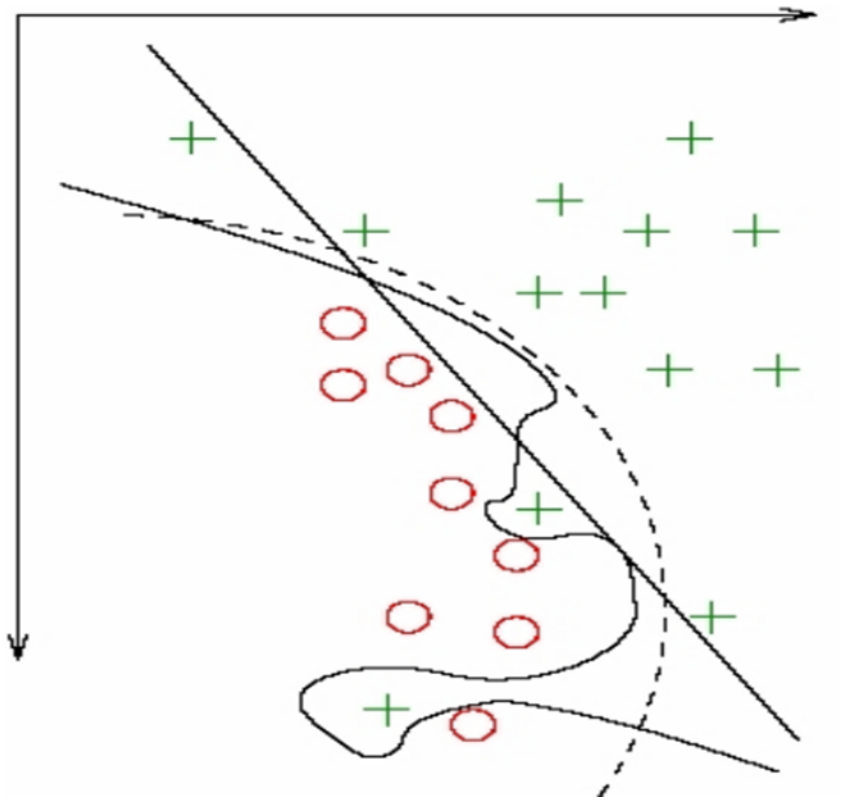


Figure7:

Rather than estimating as a value function in {-1; 1}, we search f such that = sgn(f). We then define the quantity | Yf (X) | like the margin of f in (X, Y). It's a sign of confidence. For calculations, we need:

Linear separation by SVM

- In the case where the separation is possible, one chooses the one that obtains the optimal margin.

  - The hyper plan has an equation of the form: f (x) = <w; b> + b = 0 where w is a vector orthogonal to the plane.

  - We calculate the distance between a point x and the hyperplane that is given by the formula : d (x, H) = | f (x) | / || w ||

- When the data is not separable by a plane, the constraints are relaxed by: yif(xi) 1 − i

  - The sum of all i must be minimal to cause the least error of the algorithm.

Nonlinear separation with SVM

- In this case, we use an application of Rp in H provided with a dot product, and bigger than Rp.
- We define an application k, (kernel) which verifies: (x, x') Rp Œ Rpk(x, x') = <(x); (x')>H
- The data in H are separated using the same linear hyperplane formulation (but in H!).

Part III

Detection ecosystem

6 Sources of datasets

A dataset is a set of system call data that provides accurate information about the behavior of an application for learning the normal or abnormal behavior pattern. These system call traces are collected during the normal operation of benign applications on the mobile device for normal learning on the one hand and on the other hand collected from infected processes that contain malicious behaviors

of infected applications on the mobile device. For example, all applications requests such as network communication, file management, or process-related operations must go through the kernel using the system call interface before they are executed; these data provide accurate information about the behavior of an application.

7 Environments for experimentation

The experimental environment to run malware detection tests could be a real Android device or an Android emulator. **Emulators** offer a number of advantages such as switching between API versions and screen sizes, resetting the system to clean the state automatically and root access. However, emulators present some weaknesses such as issues in cloud technologies, where 3G connections have too high latency for querying the cloud of some vendors.[22], limiting the number of apps that can be properly tested. It is therefore recommended to run the test on real devices as they offer a real user experience, the possibility to activate applications through SMS, and a convincing environment for vendors to test their software.

Server: allows you to run the discovery process outside the device on a remote server. Generally, the detection techniques that are processed remotely, are complex and require great power. This type of analysis is the simplest solution is often the most used by mobile intrusion detection systems and the most adequate for systems with limited resources since any process load will be handled remotely and not on the device However, the detection system always depends on a reliable and stable connection, otherwise the device will be almost completely vulnerable.

8 Criteria and measures of performance

In this section we shall present the criteria (requirements) that a method for detecting mobile malware is expected to have and the measures(metrics) for evaluating detection performance. The Criteria for dynamic mobile malware detection globally involves the complexity of the detection algorithm, the platform where it can be applied, and its ability adapt to new situations. Some specific criteria include:

- **Detection accuracy(rate):** This is a basic criterion for evaluating the performance of a detection method. The higher the detection rate, the better the method performs. Speed of detection: Detection should be fast and efficient. It should be real time on the device and not taken to a third party detection platform.

- **Privacy preservation:** a good method should prevent information leakage and malicious manipulations from the environment used for detection. It should not intrude in the privacy of mobile device users.

- **Ability to identify unknown apps:** A dynamic mobile malware detection method should be able to identify unknown apps and zero-day attacks.

- **Economic resource consumption.** A good detection method should consume less resource. High resource consuming methods are not good even if they produce high detection rates.

- **Choice of classification algorithms**: The efficiency and accuracy of a detection method is highly influenced by the classification algorithms it uses. It is important to properly choose or design a classification algorithm used for detection.

Some measures used to evaluate the performance of a detection method are summarised in the table below

Measures	Explanation
True Positive (TP)	Malicious programs correctly identified as malicious, i.e., True Positive = correctly identified;
False Positive (FP)	Benign programs incorrectly identified as malicious, i.e., False Positive = incorrectly identified;
True Negative (TN)	Benign programs correctly identified as benign, i.e., True Negative = correctly rejected;
False Negative (FN)	Malicious programs incorrectly identified as benign, i.e., False Negative = incorrectly rejected;
Recall: True Positive Rate (TPR)	$TPR = \frac{TP}{TP+FN}$, i.e., sensitivity or recall means the benefits we gain;
True Negative Rate (TNR)	$TNR = \frac{TN}{TN+FP} = 1 - FPR$, i.e., specificity means the costs we spend;
False Positive Rate (FPR)	$FPR = \frac{FP}{FP+TN}$, i.e., false alarm rate;
Precision: Positive Prediction Value	$P = \frac{TP}{TP+FP}$;
F-score (F-measure)	$F - score = F - measure = \frac{(1+\alpha^2)P \times TPR}{\alpha^2(P+TPR)}$, α is a pre-defined parameter;
Accuracy = Detection Rate	$Acc = \frac{TP+TN}{TP+FP+TN+FN}$;
Receiver Operating Characteristic (ROC)	Defined by FPR and TPR as x and y axes respectively, it help us determine trade-offs between True Positive and False Positive, in other words, the benefits and costs;
Area Under the Curve (AUC)	$AUC = \int_{-\infty}^{+\infty} TPR(t)FPR(t)dt.$

Figure 8: table of Measures used for detection performance evaluation

Sensitivity is a statistical measurement that stands for the proportion of correctly identified positive represented here by TPR and the hit rate. Specificity is a statistical measurement that measures the proportion of correctly identified negatives represented by TNR.

The Detection Rate and the accuracy represents the correct classification rate of all detection samples.

Precision and recall do not include all the results and samples in their formula, as such they are not adequate for showing for showing the performance of detection. They may be even contradictory to each other. To compensate for this disadvantage, F-score is calculated based on precision and recall.

Receiver Operating Characteristic (ROC) is a statistical plot that depicts a binary detection performance while its discrimination threshold setting is changeable. The ROC space is supposed by FPR and TPR as x and y axes, respectively. It helps us determine trade-offs between TP and FP, in other words, the benefits and costs. Since TPR and FPR are equivalent to sensitivity and (1-specificity) respectively, each prediction result represents one point in the ROC space. The point in the upper left corner or coordinate (0, 1) of the ROC curve stands for the best detection result, representing 100% sensitivity and 100% specificity. This point is also called perfect detection. An Area Under the Curve (AUC) is usually between 0.5-1.0, the bigger, the better the detection is.

Different measurements could be contradictory with each other. It is hard to meet with high precision and recall at the same time. To make a trade-off to balance them, F-measure i.e., F-score is often used to indicate detection performance. Based only on system calls Coupling with other features.

9 Detection approaches

Android malware detection approaches have greatly evolved from static code segment analysis to semantic analysis; from system calls to calling graphs and from codes to behaviour. This evolution is stimulated by the techniques used by mobile malware[13]

The system calls based approach is a dynamic approach that explore the patterns that a malware present during a system call to detect malware. In Android, functions of system calls mainly include hardware-related services (using a microphone, accessing a disk drive or a camera, etc.), creation of new processes, and so on [17]. Malicious applications and trusted applications have different behaviours. For instance, malicious Android applications request more permissions or access sensitive resources more frequently.[18]. It is essential to capture which system calls an application involves and the dependencies among those system calls. The system call sequences reveal the dynamic behaviours of applications and different system call sequences reflect different behaviours. Consequently, system call sequences can be used to extract features for malware detection.

In addition to the system call based approach, we have the **Anomaly-based Detection**, a classical method [13] that aims at building a model that contains the applications' normal behaviours used to classify an application's maliciousness or benignancy. In this technique, there are chances of detecting previously unseen malware accompanied by high omission on rate. In this method, a profile model of normal behaviours of the application is built according to the normal behaviors of a host and inspection applications; and the behaviour information is monitored during the execution of query samples and compared with the profile model. Here, machine learning algorithms such as artificial intelligence algorithms, and data mining methods can be applied to detect anomalies.

Again, there is a **Dynamic Specification-based Detection**, which is a special type of anomaly based detection method originating from the inspiration of law. Here authorised behaviour rules are identified by dynamically comparing observed behaviours with pre-determined authorised behaviours, called specification, of generally accepted definitions of benign activities. These techniques use a rule set of valid and legal behaviours to decide the maliciousness and benignancy of an application. The programs that violate the specifications are classified as malicious and those that don't are benign.

A part from the above approaches to detect mobile malware in general, we also have **the dynamic signature based detection** that relies on signatures and monitored packets. It uses pre-configured and pre-determined attack patterns that are given by experts to build a signature database which can be updated and used to check the maliciousness of a program. It uses information gathering during the execution of an application to determine its maliciousness. However, the polymorphic, oligomorphic and metamorphic malware issues remain a challenge to this approach.

10 Comparison of detection methods

In this section, we shall review some literature on android malware detection methods based on system calls and compare the various approaches using the criteria viewed in the above paragraph and the information presented.

Table below compares the above methods for malware detection

Ref/Year	Technique categories, classification algorithms, features and measures used, performance, comments, etc.											
	ToD	CA	Fea	P.A	R	PP	Threat	Mea	NBA	NMA	Value	Remarks
[1] 2015	SPE	NB, J48, SVM	Per, API SC	Ser	×	×	Privacy leakage	TPR, Fscore			98.4%, 0.983	A novel fusion method and good performance, heavy computation burden
[2] 2014	SIG	M	API, SC	Ser	×	×	All kinds	Acc	4800	1200	96%	Unable to list all sensitive behaviors, but high detectionaccuracy, collected data set is big.
[3] 2016	SPE	MP	SC	Har	√	×	Network attacks	Acc, AUC, FPR			≥90%, 0.997, 1.2%	Hardware architectureimplementation, flexible use with high accuracy, resource consuming, and detectionis not performed at real mobile devices.
[4] 2014	A	DCA	SC	Ser	×	×	Anomaly behaviors	Acc	50	50	98.4%	3 classifiersused for malware detectionwithcomplex classification
[5] 2013	A	RF, L, J48	Bat, SC, Per	Ser	×	×	All kinds	TPR, FPR, Acc			97.3%, 31.03, 81.25%	A generic platform to evaluate algorithms of malware detection, widely used, performance is not so good, and detection is not real-time
[6] 2016	A	M	SC	Ser	×	×	All kinds	Acc, FPR	126 187 293 437	147 195 309 483	≥90%, <4%	A hybrid and genericmethod, not real-time, high memory consumption
[7] 2017	SIG	SVM	SC	Dev	×	×	All kinds	Acc	9804	2794	99.7%	Hybrid on-devicedetectionwith high accuracy, but consume many local resources

[8] 2017	SIG	SA	RB SC	Dev Ser	×	×	All kinds	Acc			99%	Combine bothstatic and runtimefeatures to design aclient-sever detector with 99% accuracy, heavy communication burden and lack of privacypreservation in server.
[9] 2017	SIG	NB DT et al	SC	Ser	×	×	All kinds	AUC			600%	Detect malware with compressed features and work well. But results are based on two hypotheses.
[10] 2016	SIG	SVM CNN	API SC	Clo Dev	√	×	All kinds	Acc FPR			91.5% 4%	New idea to combine cloud and local detection, Real-time detectionneed good network and high-end mobile devices support.
[11]2016	SIG		SC	Ser	×	×			500	3723	99%	
18]2016	SIG	ARO W, KNN , L, NB, DT, RF,S VM	SC	Ser	×	×	All kinds	Acc	1189	1227	97,7%	
[19]2017	SIG	SVM	SC	Ser	×	×	All kinds	Acc	1189	1227	97,23%	
[20]2017			SC	Ser	×	×	All kinds	Acc FPR	570	5130	98.61% 6.88%.	
[21]2016		SVM , RF, LAS SO, RR	SC	Ser	×	×	All kinds	Acc	8371	4289	93%	

ToD: Type of Detection; **A**: anomaly-based; **SIG**: signature-based; **SPE**: specification-based. **CA**: Classification Algorithm; **N/A**: Not Available; **M**: Pattern Matching; **L**: Logistic; **RF**: Random Forest; **NB**: Naive Bayes; **SVM**: Support Vector Machine; **DCA**: Dendritic Cell Algorithm; **MLN**: Markov Logic Network; **MP**: Multilayer Perceptron; **NUE**: Neural Network; **DT**: Decision Tree; **K.m**: K-means; **NLP**: Natural Language Processing; **Gau**: Gaussian misture; **SA**: Similarity Algorithm. **Fea**: Feature used for detection; **SC**: System Calls; **Per**: Permission; **Bat**: Battery; Mem: Memory; **TG**: Topology Graph; **CC**: Covert Channel; **IF**: Information-Flow; **Net.B**: Network Behaviors; **Pac**: Packet; **Lib**: Library; **IBT**: Irrelevant Bad Terms; **RB**: Runtime Behaviors. **P.A**: Place of Analysis; **Dev**: on the device; **Ser**: server; **Clo**: cloud; **Har**: Hardware architecture. **R**: Real-time or not; **PP**: privacy preserving in out sourced/data-exchange process or not; **Threat**: the threats to overcome. **Mea**: Measures used for performance evaluation; **Acc**: Accuracy; **MSE**: Mean Square Error; **FDR**: False Detection Rate **NBA** : Number of benign Applications ; **NMA** : Number of Malware Applications;**RR**: Ridge Regularization

Tong and Yan (2016)[6] relies on dynamically collected system calls to build malware pattern set and benign pattern set based on both system calls and sequential system calls by comparing the patterns of malware and benign apps with each other. For detecting an unknown app, they use a dynamic method to collect its system calling data and compare them with both the malicious and normal pattern sets offline in order to judge the unknown app. Their approach is based on the difference of the malware and benign apps in runtime system calls. The method requires to seek more normal and malware patterns

to keep detection accuracy because new types of malware continue to emerge. Another challenge to achieve runtime malware detection with limited resources equipments.

Quan et al.(2014) [2] combines three features related to sensitive behaviour: API calls, native code dynamic execution, and system calls, for malware detection. They expressed these three features inuniform and applied a lightweight deterministic function to classify malware. Its detection rate depends on a predefined threshold. They work on 4800 benign applications and 1200 malware applications with an accuracy of 96%. However, authors do not provide details about collect and select system calls and API calls. Therefore, the contents of feature vectors are unknown. This method is built upon predefined datasets and lacks genericity.

In Ng and Hwang(2014) [4], Ng and Hwang applied Dendritic Cell Algorithms (DCA) to the collected system calls to derive the feature vectors. They run apps during 60 seconds during which they log and count invoked system calls. They either perform interactions (WI) with the application by pression screen buttons as much as possible during runtime, or track system calls with no interaction (NI). They experimented with 50 benign applications and 50 pieces of malware and obtained accuracy of 98.4% for WI and 94% for NI. They apply DCA solely on each system call although considering combination of system calls may indicate malicious behaviours.

Martinelli et al.(2017) [7] combines static and dynamic analysis methods to analyze ngrams, system calls and privileges. The static analysis is based on n-grams classification where, the frequency of opcodes is calculated from the decompiled apps, hence analyzed through a binary classifier. The framework has been tested against 2794 malicious apps re-porting a detection accuracy of 99,7% and a negligible false positive rate, tested on a set of 10k genuine apps. The dynamic analysis controls suspicious activities related to text messages, system call invocations and administrator privilege abuses.

Shina S. Et al[1] have considered Android based malware for analysis and a scalable detection mechanism is designed using multifeature collaborative decision fusion (MCDF). They consider the different features of a malicious file like the permission based features and the API call based features in order to provide a better detection by training an ensemble of classifiers and combining their decisions using collaborative approach based on probability theory. The performance of the proposed model is evaluated on a collection of Android based malware comprising of different malware families and the results show that their approach gives a better performance than state-of-the-art ensemble schemes available.

Yang Lui et al(2016) [3]proposed a hardware-enhanced architecture, GuardOL, to perform online malware detection. GuardOL is a combined approach using processor and field-programmable gate array (FPGA). their approach aims to capture the malicious behavior (i.e., high-level semantics) of malware. To this end, they first propose the frequency-centric model for feature construction using system call patterns of known malware and benign samples. Then they develop a machine learning approach (using multilayer perceptron) in FPGA to train classifier using these features. At runtime, the trained classifier is used to classify the unknown samples as malware or benign, with early prediction. The experimental results show that their solution can achieve high classification accuracy, fast detection, low power consumption, and flexibility for easy functionality upgrade to adapt to new malware samples. One of the main advantages of their design is the support of early prediction-detecting 46% of malware within first 30% of their execution, while 97% of the samples at 100% ofheir execution, with <;3% false positives

Amos B et al(2013)[5] present the evaluation of a number of existing classifiers using a dataset of thouands of rel applications, with a framework, STREAM framework, developed to enable large sclae validation of mobile malwawaremachine learning classifiers.

Wuchner et al(2017)[9] proposed the use of a compression-based graph mining approach for behavior pattern extraction. Adopting a behavior-based detection model that represents malware behavior as Quantitative DataFlow Graphs (QDFGs), they showed that patterns mined with the compression-based algorithm outperform patterns that were mined with the purely frequency-based approach in terms of malware detection accuracy. They further show that considering quantitative data flows, encoded in QDFGs, for determining graph compression levels yields better results than using graph structure properties for computing compression factors. They address the problem of finding interesting patterns in malware behavior graphs that are sufficiently discriminating to provide high detection accuracy at reasonable mining costs. In particular, we aim at a notion of pattern utility that leads to better detection results than simply considering pattern frequency as utility metric, as usually done in related work. They solution consist of adopting and modifying a well-known compression based graph

mining algorithm for QDFGs to mine discriminatory malware behaviour patterns. QDFGs model malware behavior as aggregations of quantified data flows between system entities and are induced by executed system calls. Matching the obtained patterns on QDFGs of known malware and benign software they train a supervised machine learning classifier that they use for classifying unknown malware samples. They are the first [9] to use compression-based graph mining for behavior-based malware detection using quantitative data flow information and show that patterns obtained using compression based mining lead to 600% more accurate detection results than patterns obtained with a commonly-used frequency based mining algorithm.

Isohara et al. (2017)[15] proposed a method including a log collector and a log analyser. The first component collects all system calls from the Kernel and filters events related to the running application. The system calls related to process management and file I/O activity.The second component matches activities with signatures described by regular expressions to detect a malicious activity. Collecting data is made on an Android 2.1 based modified ROM image and analysing and detection data is made on a server. This method uses signature-based detection reputed ass less efficient. False positive rate is higher because the system calls collected in this work can be misclassified as malicious. They do not consider system calls generated after interactions with the application.

Xiao1, Xianni Xiao1, Yong Jiang1, Xuejiao Liu and Runguo Ye (only SC) [18] {Apk-downloaderchrome pour télécharger les APK}[ ] track the system call sequence of each application in a real Android device. They use two different models, the frequency vector and the co-occurrence matrix, to characterise the system call sequence. Finally, they identify the malicious applications with machine learning algorithms. This work generates different kinds of events for applications using monkey. However, results from this work are related to the collected samples. Unknown ones could provide misclassifications. 1189 benign applications and 1227malicious applications. Detection rate is 97.7 per cent. They track system calls of each application with strace - process ID and its descendants process IDs (process grained)

Android version 4.0.4 AROW, KNN, logistic, naive Bayes,decision tree, random forest and SVM

CSCdroid: Accurately Detect Android Malware via Contribution-Level-Based System Call Categorization[19]

CSCdroid proposed a novel approach based on the contribution of system calls for malware identification. The contribution concept depends on the occurrence times of a system call in the SC sequence from benign (malware) samples and the number of all the system call sequences from benign (malware) applications. Authors use Markov chains to build feature vectors from the modified sequence of calls, that are transmitted to SVM for classification.

Support Vector Machine (SVM)

They utilize the SC sequence, the SC invocation list from the start up to the exit of one app, to investigate its real purposes.

They use Monkey to inject 1000 events strace to trace all 196 SCs in Android 4.0

1189 benign applications and 1227 malicious applications

we exploit the Markov chain to construct the feature vectors and apply SVM classifier to identify malware. CSCdroid gets TPR of 97.23%

Vidal et al.[Malware Detection in Mobile Devices by Analyzing Sequences of System Calls] [20] monitored only sequence of system calls during the app start-up for the identification of malware.Performance decreases with short sequences.

Client serverStrace 570 legitimate samples and 5130 malicious samplesTPR rate is 98.61% and the FPR is 6.88%.

Marko et al [21] : Maline is an Android-based emulator that tracks and logs system calls during runtime to identify malware samples. It encodes the generated log files into feature vectors for further classification. Authors process feature vectors with the number of occurrences of a system call during an execution of an application and distances between the calls of two system calls in a sequence. They build a feature vector with all the system calls available in the Android operating system for a given processor architecture. This method is coarse-grained resource consuming and manipulates useless system calls. They ignore the fact that malware could potentially detect it is running in an emulator, and alter its behaviour accordingly. Support vector machines (SVMs) , random forest (RF), LASSO, and ridge regularizationAndroid 4.4.3 KitKat release, which utilizes Android API version 19They consider system call names and their order in sequences.Monkey tool [25] as our internal event generatorstrace tool.QEMU4289 malicious and 8371 normal; detection accuracy of 93% with a 5%

benign application classification error,All x86 system calls ->360system calls32-core machineswith 128 GB of RAM running Ubuntu 12.04 Client serverx86Android emulator

Part IV
Discussions and improvements

1. Most of the methods examined perform feature analysis and unknown software detection at a third party, such as a server or a cloud, which could cause device user privacy disclosure since collecting device data for malware detection could intrude the device user's privacy. None of existing work considered protecting these data when outsourcing. Hence, protecting user privacy in malware detection performed at the cloud or the third party server is still an open and untouched issue.

2. We noticed that the existing methods are hardly deployed at the mobile device, thus hard to support real-time detection. The main reason is mobile malware detection normally requests big data collection and pro-cessing. But due to the limitation and restriction of computing resources, processing capability and memory storage at the mobile device, it is still not practical to implement most of existing methods at the device. Some existing methods try to realize real-time detection with the assistance of cloud or specilized devices, but they need further improvement. Idealy, real-time malware detection should be realized at the mobile device in an efficient way.

3. We also observe that ROC, F-measure and AUC are seldom applied as the measures to demonstrate the performance of malware detection. The measures used for evaluating the detection performance in many existing studies cannot comprehensively reflect each angle of the quality of malware detection. Other evaluation methods/measures could be needed, e.g., a game theoretical analysis method, to evaluate other quality attributes (e.g., applicability) of a detection method.

4. In addition, the features used for malware detection are not comprehensive, which implies that some threats or attacks may not be detected since they are only reflected by some unused features. We are still confronting challenges to overcome the above problems.

5. Also the number of benign to malware applications used for testing is still very limited a few hundreds [2] and thousands [11], the high degree of accuracy may be due to the fact that the data set under test is very limited. We look forward to methods that take a great number of applications that reflect the android ecosystem.

6. The most used features with system calls in most peppers is the permission based method.

Part V
Open issues and future research trends

11 Open issues

A number of open research issue s can be figured out from the above research review. These include:

1. Difficulties in obtaining real-time detection. Most detection methods are not real time in terms of app runtime behavior monitoring, talkless of real-time detection. GuardOL [11] designed and implemented a hardware-based architecture for malware detection. It was said as a kind of real-time detection method, but actually not. It monitors apps' system calls and constructs a frequency-centralized model to identify malware on specialized FPGA. But it fails to reveal all system calls while extracting features. In practice, we need real-time malware detection during app execution, especially for finding out some intrusions or attacks happening at app runtime.

2. The semantics of parameters to identify and evaluate the maliciousness of a system call remains a challenge that can be addressed

3. There is also a difficulty to have all the real time system calls corresponding to a user's action. If a tool could be developed to track all these calls in real time it could be great.

4. Again, system calls evolve across Android versions, making it difficult for version based detection methods to cut across.

5. Besides system call, detection methods that makes use of library calls and method calls can be put in place.

6. The literature we have seen so far does not consider the sequence order in which the system call appears in a system call sequence. A method that takes into consideration this order may yield better results.

7. The co-occurrence matrix can only model behaviour between two system calls. A method that takes into account more than two system calls can be more appropriate to exploit dynamic behaviour amongst system calls.

8. A knowledge base with an expert system can be built to combine the existing features can bring about a better detection system.

9. There is a need to protect information privacy if private data are outsourced and malware detection is conducted at a distrusted third party. Many cloud-based methods were proposed for mobile malware detection due to constrained resources of mobile devices. Many detecting schemes choose to identify malware in the cloud. This could intrude user privacy since the cloud cannot be fully trusted. However, none of existing work provided privacy protection during the process of mobile malware detection.

10. Big data introduces special challenges in malware detection, especially dynamic real-time detection. There is a rapid growth of malware, yet most existing work normally used a small sample set to evaluate the performance of detection. Thus, the performance evaluation results were not so convinced, which is what we should improve in the future research. But applying big data to do performance test requests big data collection, especially a big set of app samples. On the other hand, some detection methods need to process big data. Both of them introduce special challenges about big data process in terms of mobile malware detection. How to efficiently process and analyze big data in order to realize real-time malware detection in a comprehensive way is still an open issue worth our efforts. Obviously in the era of data explosion, we need to explore how to manage and process big data for efficient malware detection.

Thus research on dynamic mobile malware detection in general and system call based detection still needs to improve significantly. There is therefore the need to rapidly solve the above open issues and improve upon the existing methods to meet practical demands.

12 Future research trends

Future research trends are directly linked to open issues.

1. High detection accuracy and efficient detection algorithms are always highly expected.

2. Information privacy protection should be investigated with regard to cloud-based mobile malware detection. Due to the constraint resources of the mobile devices, cloud-based detection becomes an inevitable tendency. But how to avoid the unexpected leakage of user privacy in the process of collected data for malware detection is a practical and crucial issue. Privacy-preserving mobile malware detection over the cloud is an interesting and significant research topic. In this research, secure transmission protocols, safe data storage with flexible access control and most importantly secure data processing should be studied.

3. Real-time detection with lightweight detection algorithms is another interesting research direction. Malware detection by monitoring behaviors, evidence and features can identify zero-day attacks and detect malware after malware infection. But there is a serious time delay. Most existing methods are either server-based or cloud-based systems. They are pretty heavy and complicated with a long time delay. The methods that are both lightweight and efficient thus can be installed in the mobile devices to realize real-time detection, are highly expected. This kind of methods can find newly generated malware initiated at app runtime during its execution. Although designing such a detection method is not easy, it is a very promising research direction.

4. The huge number of malware samples and big size of collected data for malware detection cause a big data problem, which challenges the detection and forces it to be much efficient to handle big data. The speed of malware growth has never slowed down. The sample database is becoming enormous. So finding a way to dramatically and efficiently reduce the sample space, and effectively detect malware is urgent. To some possible extent, distributed detection systems and cloud based solutions can make this problem easier. Besides, data mining and fusion methods and some other strategies used in big data processing can also be applied in solving the big data issues in this research field.

References

1. Sheen, S., Anitha, R., Natarajan, V.: Android Based Malware DetectionUsing a Multifeature Collaborative Decision Fusion Approach. Neurocomputing. 151, 905-912 (2015)

2. Quan, D., Zhai, L., Yang, F., Wang, P.:Detection of Android Malicious Apps Based on the Sensitive Behaviors. In: 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). pp. 877-883. IEEE, Beijing (2014)

3. Das, S., Liu, Y., Zhang, W., Chandramohan, M.:Semantics-Based Online Malware Detection: Towards Efficient Real-Time Protection Against Malware. IEEE Trans.Inform.ForensicSecur. 11, 289-302 (2016)

4. Ng, D., Hwang, J.: Android Malware DetectionUsing the DendriticCellAlgorithm. In: 2014 International Conference on Machine Learning and Cybernetics (ICMLC). pp. 257-262. IEEE, Lanzhou (2014)

5. Amos, B., Turner, H., White, J.:Applying Machine Learning Classifiers to Dynamic Android Malware Detection at Scale. In: Wireless Communication and Mobile ComputingConference (IWCME). vol. 14, pp. 1666-1671. IEEE, Nicosia (2013)

6. Tong, F., Yan, Z.: A hybridApproach of Mobile Malware Detection in Android. Journal of Parallel and DistributedComputing. Elsevier, in press (2016)

7. Martinelli, F., Mercaldo, F., Saracino, A.: BIRDEMAID: A HybridTool for AccurateDetection of Android Malware. In:Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. pp.899-901. ACM, Abu Dhabi (2017)

8. Sun, M., Li, X., Lui, J., Ma, R. Liang, Z.: Monet: A User-OrientedBehavior-Based Malware VariantsDetection System for Android. IEEE Transactions on InformantionForensics and Security. 12, 1103-1113 (2017)

9. Wuchner, T., Pretschner, A.:Leveraging Compression-based Graph Mining for Behaviorbased Malware Detection. IEEE Transactions on Dependable and Secure Computing. inpress (2017)

10. Hung, S., Tu, C., Yeh, C.: A Cloud-Assisted Malware Detection Framework for Mobile Devices. In: 2016 International Computer Symposium. pp. 537-542. Merida (2016)

11. Das, S., Liu, Y., Zhang, W., Chandramohan, M.: Semantics-Based Online Malware Detection: Towards Efficient Real-Time Protection Against Malware. IEEE Trans.Inform.Forensic Secur. 11, 289-302 (2016)

12. F. Tchakounte, P. Dayang : System Calls Analysis of Malwares on Android. International Journal of Science and Technology Volume 2. No.9 (2013)

13. Zheng Yan, : A survey on dynamic mobile malware detection . Article in Software Quality Control (May 2017)

14. F. Tchakounte: A malware Detection System for Android. Thesis September 2015.

15. Isohara et al : Kernel-based Behavior Analysis for Android Malware Detection.EEE Computer SocietyWashington, DC, USA©2011

16. Canfora et al : Detecting Android Malware using Sequences of System Call. Bergamo, Italy August 31, 2015,

17. System call https://en.wikipedia.org/wiki/System_call> (accessed 18.04.17)
18. Xiao et al: Identifying Android malware with system callco-occurrence matrices.transactions on emerging telecommunications technologies.Trans. Emerging Tel. Tech.2016;27:675–684.Published online 3 February 2016 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/ett.3016
19. Shaofeng Zhang, Xi Xiao: CSCdroid Accurately Detect Android Malware via Contribution-Level-Based System Call Categorization.2017 IEEE Trustcom/BigDataSE/ICESS
20. Vidal et al: Malware Detection in Mobile Devices by Analyzing Sequences of System Calls.
21. Marko et al: Android Malware Detection Based on System Calls
22. Hendrik Pliz; Building a test environment for Android anti-malware tests. Magdeburg 2012
23. Davy Leggieri: Les threads composants une application Android
24. Shauvik et al : Automated Test Input Generation for Android.