Detecting Malicious Applications using System Services Request Behavior

Majid Salehi
imec-DistriNet, KU Leuven
majid.salehi@cs.kuleuven.be

Morteza Amini
Sharif University of Technology
amini@sharif.edu

Bruno Crispo
imec-DistriNet, KU Leuven
Trento University, Italy
bruno.crispo@cs.kuleuven.be

ABSTRACT

Widespread growth in Android malware stimulates security researchers to propose different methods for analyzing and detecting malicious behaviors in applications. Nevertheless, current solutions are ill-suited to extract the fine-grained behavior of Android applications accurately and efficiently. In this paper, we propose ServiceMonitor, a lightweight host-based detection system that dynamically detects malicious applications directly on mobile devices. ServiceMonitor reconstructs the fine-grained behavior of applications based on their interaction with system services (i.e. SMS manager, camera, wifi networking, etc). ServiceMonitor monitors the way applications request system services in order to build a statistical Markov chain model to represent what and how system services are used. Afterwards, we use this Markov chain as a feature vector to classify the application behavior into either malicious or benign using the Random Forests classification algorithm. We evaluated ServiceMonitor using a dataset of 8034 malware and 10024 benign applications and obtaining 96.7% of accuracy rate and negligible overhead and performance penalty.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; *Mobile platform security*.

KEYWORDS

Operating System, Android, Malware, Behavior Detection

ACM Reference Format:
Majid Salehi, Morteza Amini, and Bruno Crispo. 2019. Detecting Malicious Applications using System Services Request Behavior. In *16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous), November 12–14, 2019, Houston, TX, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3360774.3360805

1 INTRODUCTION

Android is the most popular smart phone OS currently being used. Gartner [1] reported that 88% of smart phone sale in the second quarter of 2018 were Android based devices. Android has brought about many new applications, which have resulted in a complete different level of experiences for the end user. However, it has created concern as to the security and privacy of the information (i.e. contacts, geographical locations, photos, and etc.) being used/shared with new applications. As a result, and due to its popularity, Android OS has been a major target for new malware. In fact, it has been reported by AV-Test [2], that over 99 percent of new malicious applications targeting mobile devices are aimed at Android devices.

There have been many approaches [3–13] proposed to combat the rise of malware in Android devices. These techniques vary in their approaches to the problem; techniques that operate outside the device, such as Google Play Protect [3], to applications that operate on the end-user device [6, 11] and provide a malware detection service. It is important to note that these different approaches are complementary and it is a good example of defense in depth. This could be explained by the fact that end-user detection is bound by limited resources available on the device, but more importantly, market based techniques suffer from the fact that not all applications are downloaded from a single market. In fact, there are multiple alternative markets to Google play, such as Amazon app store, SlideME, and Samsung Galaxy Apps. This highlights the importance of employing also on-device malware detection techniques.

Detection techniques that run on-devices are limited by resources on the device as well as the user experience, i.e. users don't want to experience delay in app interactions due to the detection mechanism. Generally, these techniques operate by collecting behaviors, as a set of features, of the application being analyzed and then decide on the nature of the application, i.e. benign or malicious. The analysis could be done either statically or dynamically.

Static techniques analyze the application bytecode with near-complete coverage, considering all execution paths, even if a part of the program never executes. These techniques are vulnerable to transformation attacks [14] and could be evaded by traditional obfuscation techniques, such as Java reflection and bytecode encryption, or newer runtime-based obfuscation techniques [15]. Furthermore, malicious behaviors may be implemented in native codes [16], which are usually not analyzed, or they might be hidden and triggered at run-time through additional codes loaded dynamically from external sources [17].

Alternatively, dynamic analysis techniques observe the behavior of applications at run time. But as Zhang et al. [18] reported, most of these techniques which operate on end-user devices, consider the behavior of applications at the system call level, i.e. system calls executed and the order of their execution. In fact, given Android OS architecture, techniques based on the system call tracing obtain an incomplete view of the behavior of the binary being analyzed. This becomes clear, when one considers that in Android OS, applications are not able to directly access system resources

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MobiQuitous, November 12–14, 2019, Houston, TX, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7283-1/19/11...$15.00
https://doi.org/10.1145/3360774.3360805

(e.g. SMS, camera, microphone) through system calls at the kernel level. Instead, Android framework provides specific *system services* at a higher semantic level than system calls in order to manage communications among system resources and applications. Hence, traditional dynamic methods, which are based on analyzing system calls, are less effective in malware detection, as they reconstruct behaviors of applications based on a number of intercepted system calls which contain no information about system resources being accessed.

Considering above observations and taking into account issues and limitations of static analysis techniques, we propose a novel *System Service Use Analysis* technique to capture and analyze the fine-grained behavior, i.e. at the system services level, of applications with the aim of detecting malicious applications. We build a statistical model to represent what and how system services are used to access system resources. Specifically, we model sequences of requested functions from system services as Markov chains, and use them to extract features and perform classification.

In fact, ServiceMonitor systematically analyzes and models system services use at multiple levels of semantics regardless of whether it is from Java or native code, and classifies Android applications as malicious or benign directly on mobile devices. In summary, this paper makes following contributions:

- *System service use analysis.* We propose a systematic system services use analysis technique, which automatically and seamlessly models the state transitions achieved by functions requested from system services as a Markov chain; aim at representing the application behavior by its pattern of accesses to system resources.
- *Effective Android malware detection.* We developed a dynamic Android malware detection framework, named ServiceMonitor, to reconstruct the behavior of applications based on system services use analysis technique with the aim of identifying malicious behaviors as well as malware. ServiceMonitor is capable to detect Android malware with high accuracy and few false positives.
- *Lightweight detection procedure.* With taking into account the limitation of resources in mobile devices, ServiceMonitor operates with a low performance overhead and using limited energy.

In the remainder of this paper, we first survey a number of related work in Section 2. Then we describe inter-/intra-process communication procedure in Android OS in Section 3. Then the proposed system service use analysis technique and ServiceMonitor detection method are described in Section 4. After that we present the implementation details and evaluation results in Section 5 and Section 6 respectively. We also discuss some related issues on Android malware detection and limitations of ServiceMonitor in Section 7 and conclude the paper in Section 8.

2 RELATED WORK

There exist many techniques in the literature for analyzing and detecting Android malware. These techniques could be categorized based on how the analysis/detection agent is deployed as: **i) Emulator-based techniques**, such as [19, 20], which are based on software virtualization and emulation. These systems suffer from several fundamental limitations, i.e. performance penalties, transparency issues, and special software/hardware requirements, for deploying on end-user devices. **ii) Cloud-based techniques**, such as [12], which collect information from end-user device and then aggregate and analyze the information on the cloud. The effectiveness of such frameworks relies on the reaction of end users when they are asked to send recorded logs from applications to an external server. **iii) Host-based techniques**, which are deployed wholly on the mobile device given the performance constraints, without requiring emulators or cloud based analyzers. These systems can be categorized into static and dynamic approaches.

2.1 Static Analysis and Detection

There have been a number of static learning based studies in which features are defined and selected based on some malicious code patterns and heuristics. For instance, DroidSieve [8] and DroidDet [10] are two similar approaches that extract detection features from applications disassembled codes and manifest files as much as possible. These two approaches hold the occurrence of sensitive API calls in their feature sets beside other information such as requested permissions and names of application components, i.e. activities, services, broadcast receivers, and content providers. These approaches also differ from each other in how to structure features extracted from applications. ICCDetector [9] is a static based method that extracts ICC (Inter-Component Communication)-related features that hold interactions within or cross applications components, and then leverage machine learning techniques to perform classification. As noted briefly in introduction and stated in [14] and [15], static analysis and detection techniques are thought to be insufficient to detect malware variants generated by transformation and obfuscation attacks. Also, most of the current static solutions are ill-suited to analyze additional codes loaded dynamically from external sources [17] as well as native codes [16].

2.2 Dynamic Analysis and Detection

On the other hand, there have been a number of proposed works [5, 11, 21–24] in which applications are analyzed dynamically using extracted system calls in order to detect malware at runtime. However, as Zhang et al. mentioned in [18], all system call based systems share one fundamental limitation. Due to the missing semantic view of accesses to system resources, their analysis is ineffective to demonstrate the fine-grained and accurate behaviors of Android applications. To solve this limitation, Sun et al. [13] proposed Patronus, which focuses on fine-grained behaviors of applications. Patronus is a host based intrusion prevention system for Android devices. Patronus considers a database of manually crafted malicious policies from known malware samples and calculates the similarity score between transaction footprint extracted from running applications on user's device and malicious transactions, which are recorded in the policy database, aim at preventing malicious intrusions and detecting malware at run-time. But Patronus depends on experts to define security policies and rules to cover malware misbehaviors. Hence, due to the known malicious policies in its database, it could not be effective and suited for detecting unknown and zero-day malware families.

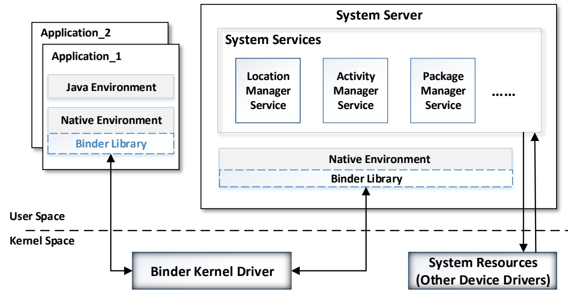

Figure 1: Accessing to system resources and interactions between applications components through Binder transactions.

In what follows, and based on noted limitations of previous related works, we propose an effective dynamic detection method for malware detection on end-user devices.

3 INTER-/INTRA-PROCESS COMMUNICATION

As illustrated in Figure 1, in Android security architecture, each application is isolated in a separate sandbox, and direct access from applications to system resources (i.e. SMS, GPS, address book, etc.) is prohibited. Instead, access to system resources is handled through system services implemented in the framework layer of Android architecture. Furthermore, alongside requesting system resources in Android OS, interactions with other apps and with other components of the same application are performed through system services. For example, sending data between different components of an application use Activity Manager system service via Binder component.

System services, similar to other processes, are run in a sandboxed environment and with a distinct system identity, i.e. Linux UID and GID, from other processes. Applications access system services through Binder component. Binder component is deployed at two levels, Binder Framework and Binder driver. The former is a user-level library named libbinder which is loaded into the most Android applications and used with the aim of marshalling requests and sending them to system services via an specific system calls named ioctl. On the other side, system services use libbinder library for unmarshalling requests and creating respond objects. The second level, Binder driver, controls all processes communications in the kernel level. In other words, libbinder, by calling ioctl system call, sends required services and requests to Binder driver, then it drives them to the targeted system service.

4 OUR APPROACH

Android applications require access to different system resources to operate properly. Where such operations could be benign as advertised by the application creator, or malicious; although appears to be benign by the client. In both scenarios system resources are accessed, but the type of resources and the order of accesses are different.

Malicious behaviors that are frequently observed from Android malware have been widely surveyed in the research literature [25–27] and malware reports [28]. Based on those studies and the analysis that we have done on both benign and malicious samples, we believe that a fine-grained behavioral model could be constructed through observing accesses to a dozen of system services and their functions implemented in the framework layer of Android OS. Furthermore, and based on Android system documentation [29], system services could be categorized into six broad categories. These categories and services that they cover are shown in Table 1.

In what follows, we discuss details of the detection process in which “ServiceMonitor" receives execution traces, i.e. functions requested from system services, of an application and then models the behavior of the application as a Markov chain. Finally, a feature vector is generated from Markov chain model and fed to a binary classifier to determine the nature of the given application as either malicious or benign. Fig. 2 represents the mentioned process beside the architecture of the proposed monitoring system.

4.1 Monitoring of Applications Service Requests

As noted earlier, all transactions to/from each application, i.e. including those appears between the application and other applications or between the application and system services, would be only possible through Binder library in Android security model.

Binder library itself is divided into two segments:

i) a user-space shared library called *libbinder.so* and
ii) a kernel-level driver.

libbinder.so is tasked with receiving requests from the user-space process and marshalling them into a *Parcel* object which is then passed to the kernel-level Binder driver for further processing, given the permission granted to the application. More specifically, *ioctl* system call is employed by applications to make requests to Binder library.

In order to obtain detailed information about system services requested by each application, we implemented a kernel module with which *ioctl* system calls are intercepted and unmarshaled (i.e. parsed). More details on the implementation are provided in Section 5. The architecture and overall design of ServiceMonitor is depicted in Fig. 2. ServiceMonitor kernel module is split into hooking and unmarshalling components for obtaining *ioctl* system calls and unmarshalling them respectively. We should note that the unmarshalling procedure, which is introduced in the following, is based on previous works [19, 30, 31] that present the structure of *ioctl* system calls and architecture of Binder transactions. *ioctl* system call has the following syntax:

ioctl(Driver_fd, BINDER_WRITE_READ, &bwr);

Since we aim at intercepting Binder transactions, */dev/binder* is the only considerable value for us that determines the file descriptor of Binder devices as the first *ioctl* argument. *BINDER_WRITE_READ* command is the basis for all IPC operations. Thus, we consider that as the request code that should exist in the intercepted *ioctl* system calls.

The last and the most important argument of *ioctl* system call is a pointer to a struct of the type *bwr* (short for *binder_write_read*). As illustrated in Fig. 3, this data structure contains a pointer to a valuable

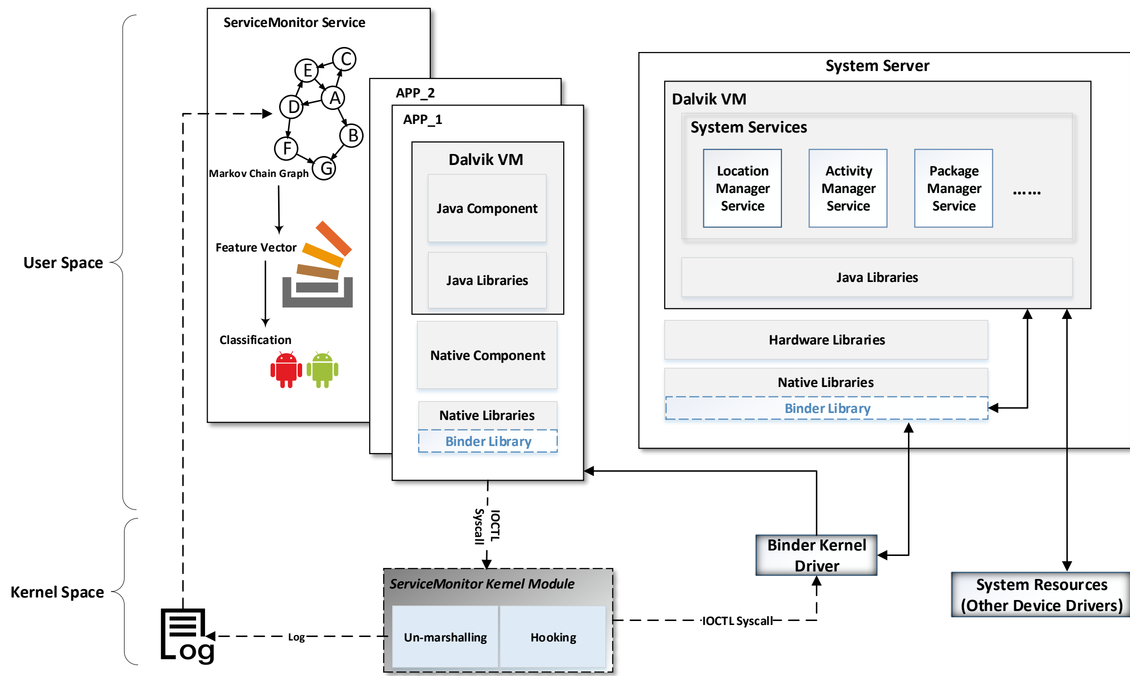

Figure 2: Architecture and detection process of ServiceMonitor.

buffer named *write_buffer*, which holds the type of transactions and respected parameters. Due to the fact that we would like to intercept Binder transactions, *BC_TRANSACTION* is the only one of these transaction types that is of interest to our work. At the next level and in Binder transactions, we are dealing with a data structure named *binder_transaction_data*. As depicted in Fig. 3, this data structure contains some valuable attributes that could be used for extracting system services, e.g. com.android.internal.telephony.ISms, and corresponding functions, e.g. sendText, that are requested through the invocation of this *ioctl* system call. *code* attribute is the code of the requested function, which is implemented in the destination system service and is required to be executed by the source application. *buffer-ptr* is a pointer to *Parcel* object that we would like to unmarshal. There is a 16-bit Unicode string named *InterfaceToken* at the start of every *Parcel* object structure. *InterfaceToken* determines the name of the system service that is considered as the server for the application request.

Finally, requested functions and system services are recorded in a chronological order as behavioral features and delivered to ServiceMonitor service for modeling and classification. Owing to the fact that these characteristics of Android OS have stabled during all version releases of Android, our IPC dissecting procedure is a portable way to reconstruct application behaviors, independent from the internal complexity of applications.

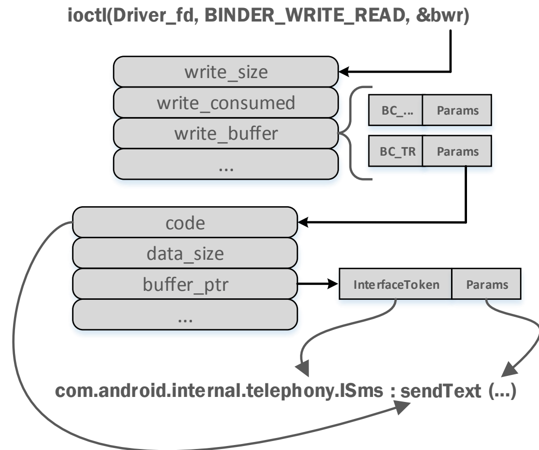

Figure 3: Dissecting *ioctl* system call. ServiceMonitor extracts the target system service (e.g. com.android.internal.telephony.ISms) and requested function (e.g. sendText) by dissecting *ioctl* system calls.

4.2 Markov-chain Modeling of Application Behaviors

ServiceMonitor employs Markov chains for modeling application behaviors. A Markov chain could be described by a weighted directed graph. In other words, in this representation there is a set of nodes that belong to different states, and a set of edges between

Table 1: System services could be categorized into 6 categories. The information noted in this table was obtained from documentations available at developer.android.com

Service Category	Interfaces	Functionalities
Telephony Manager	ISms IPhoneSubInfo ITelephony ITelephonyRegistry	Sends text messages, retrieves phone numbers, retrieves the unique device ID (e.g. IMEI), retrieves the serial number of the ICC, retrieves the unique subscriber ID (e.g. IMSI), retrieves the software version number of the device (e.g. IMEI/SV), retrieves the network type for data transmission, retrieves the current active phone type (e.g. PHONE_TYPE_CDMA, PHONE_TYPE_GSM), listens to the phone state changes
Location Manager	ILocationManager	Retrieves the last known location, registers for location updates, retrieves the list of the names of LocationProviders that satisfy the given criteria, retrieves the name of the provider that has the most compliance with the given criteria
Network Manager	IConnectivityManager IWifiManager	Retrieves the current proxy settings, retrieves the connection status information of a particular network type, retrieves the details of the current active default data network, retrieves the connection status information of all network types supported by the device, retrieves the dynamic information about the current Wi-Fi connection, retrieves the Wi-Fi enabled status
Activity Manager	IActivityManager	Starts a service, Stops a service, Resumes an activity, Idles an activity, Gets the list of running application processes, Checks permissions, Retrieves the memory information, Registers for Intent broadcasts (e.g. Boot_Completed), Broadcasts Intents, Gets a content provider, Removes a content provider, Starts an activity, Pauses an activity, Finishes an activity, Gets services, Unregisters Intent receivers, Gets the orientation, Sets the orientation, Kills a list of processes, Gets task id of an activity, Gets the sender of a given Intent
Package Manager	IPackageManager	Retrieves the list of all packages installed, retrieves information about a particular package/application, retrieves the names of all packages that are associated with a particular user id, retrieves information about an application package installed on the system, retrieves information about a particular activity class, retrieves all activities can be performed for the given intent, checks whether a particular package has been granted a particular permission, checks for the presence of the given feature name in OS
OS Related Activities	IPowerManager IServiceManager IMountService	Retrieves the overall interactive state of the device (i.e. actual state of the screen), acquires the wake lock and forces the device to stay on, releases the wake lock, retrieves an existing service with the given name, retrieves the state of a volume via its mount point, retrieves the list of all mountable volumes

them weighted with the probability of transition from each node to another one.

In our proposal, for each application, potential dependencies between states obtained by calling functions of system services are represented as a Markov chain. In other words, ServiceMonitor generates a complete weighted directed graph in which each vertex φ corresponds to a state obtained by calling a function of a system service (using *ioctl* system call). For each function F_x of system services, we take an abstract state, denoted by $State(F_x)$, representing the application state after calling the function. In this way, the generated graph has $|\varphi|^2$ edges and each edge has a weight; representing the probability of the corresponding state transition.

In this modeling scheme, weights of edges are obtained by analyzing sequences (traces) of requested functions from system services; which are logged by the monitoring system. The distance between two requested functions F_i and F_j in a sequence σ (where i and j are the indexes of them in the sequence) is defined as follows:

$$d(F_i, F_j) = |j - i|$$

In fact, the distance value $d(F_i, F_j)$ can determine the potential relationship, i.e. data or control flow, between a pair of requested functions F_i and F_j in a sequence σ. In other words, two requested functions that are closer to each other have more contribution on the weight of edge between their corresponding states in Markov chain graph.

Now, we define P_{xy} as the probability of transition from state s_x to s_y (the weight of the directed edge (s_x, s_y)) as the following, where $State(F_z)$ determines the abstract state of the application after receiving the requested service function F_z.

$$FV_{xy} = \begin{cases} 0 & \text{, if } x = y \\ \displaystyle\sum_{\substack{i<j\leq|\sigma|, \\ s_x=State(F_i), s_y=State(F_j), \\ \nexists h, (i<h<j \,\wedge\, s_x=State(F_h)}} \frac{1}{d(F_i, F_j)} & \text{, otherwise} \end{cases}$$

$$P_{xy} = FV_{xy} / \sum_{1\leq l\leq|\sigma|} FV_{xl}$$

Note that in this paper, for the sake of simplicity, we label the state of an application after receiving a requested service, by the name of the requested function from system services, e.g. see Fig.

4; which is Markov chain model of the example represented in the next section. The pseudo-code of building Markov chain graph is represented in Procedure 1.

Procedure 1 Building Markov chain graph

Input: system_service_trace, system_service_list
Output: probability_matrix

```
 1: size=len(system_service_list)
 2: Declare Integer FV[size][size]
 3: j=len(system_service_trace)
 4: for i = 0 → j − 1 do
 5:     line=system_service_trace[i]
 6:     index=system_service_list.index(line)
 7:     append index to map_list
 8: end for
 9: k=len(map_list)
10: for i = 0 → k − 1 do
11:     for j = i + 1 → k − 1 do
12:         if map_list[i] ≠ map_list[j] then
13:             FV[map_list[i]][map_list[j]]+=(1/(j-i))
14:         else
15:             Break
16:         end if
17:     end for
18: end for
19: for i = 0 → size − 1 do
20:     S=SUM(FV[i][:])
21:     for j = 0 → size − 1 do
22:         FV[i][j]=FV[i][j]/S
23:     end for
24: end for
25: Return FV
```

4.3 Feature Extraction and Classification

The final step for determining the nature of an application is extracting a feature vector and feeding the vector into a machine learning algorithm for classification of the application behavior. To this aim, we take weights of $|\varphi|^2$ edges of the Markov chain graph, which are generated from logged sequences of requested functions from system services, as a 1D feature vector $fv = [f_1, f_2, ..., f_{|\varphi|^2}]$, where f_i is equal to P_{km} so that $i = (k-1) \cdot |\varphi| + m$.

Then we employ Random Forests algorithm for classifying the application behavior to either malicious or benign using the extracted feature vector. To train the classification algorithm, we can use samples of known malware and benign applications. Section 6 describes the training, evaluation, and feature reduction process in more details.

Example: Consider an application that requests following functions respectively: *getSubscribedID, requestLocationUpdates, sendText, requestLocationUpdates, sendText.* Following the proposed approach, we monitor and extract these functions requested from system services by our proposed IPC dissecting procedure and model them in a Markov chain graph similar to the Fig. 4. Then, we extract the specified feature vector as [0, 0.64, 0.36, 0, 0, 1, 0, 1, 0] from Markov chain model of application behavior for classification purpose.

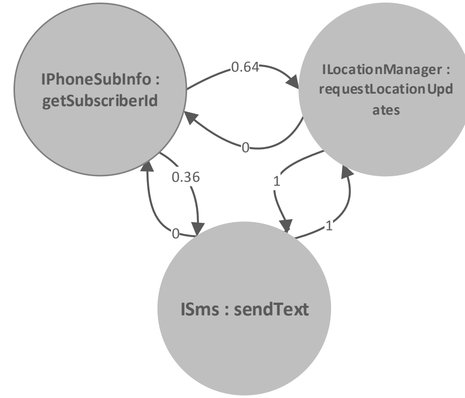

Figure 4: Markov chain model of an application behavior.

5 IMPLEMENTATION

We implemented our proposed method in a system named ServiceMonitor. The main module is a kernel module, which is implemented in *C* and is tasked to monitor applications running in the device simultaneously and log functions requested from system services by each application in their chronological order, i.e. based on their timestamps. More specifically, the developed kernel module intercepts *ioctl* system calls and dissects them to extract functions requested from system services. This is done by rewriting the address of *ioctl* function implementation in *system_call_table.*

system_call_table is implemented in kernel level and is used for organizing system functions and quick access to them. Hence, at first, we need to obtain the address of *system_call_table* from the *vector_swi* handler. Indeed, by using the technique proposed in [32], we are able to obtain the address of *system_call_table* in all versions of Android OS in a similar fashion. After that, we rewrite the address of *ioctl* function and redirect all Binder transactions to our unmarshalling component. Finally, after logging and dissecting Binder transactions, we redirect them to the original *ioctl* function for continuing the normal execution of the application.

Collected features are then processed using a second module, implemented in Python, with which the requested system services are transformed into Markov chain representation discussed earlier. Lastly, an implementation of the classification algorithm, written in *R* language [33], is used for distinguishing the malicious applications from the benign ones. Introduced modules are collected in a standalone application, which could be deployed on an end device with the aim of employing a detection model, which is trained offline, and classifying running applications as either malicious or benign.

In the next section, details on automatic execution of applications in the training phase of ServiceMonitor are presented.

5.1 Automatic Execution of Applications

In order to train the detection model of ServiceMonitor, applications, which are included in the prepared dataset, were installed on an

unmodified Android version 6.0 (Marshmallow version), which was deployed upon a specific virtual machine called VirtualBox [34] for executing and tracing them simultaneously. As mentioned earlier in this paper, the tracing of applications behaviors with the aim of extracting features are done through ServiceMonitor kernel module.

In order to process the large number of applications in the dataset, we automated the process of installation, execution, interaction, and data collection. This was done by installing each application in a clean state of the OS automatically by using *adb* tool and simulating end users' activities and interactions. We leveraged *MonkeyRunner* tool [35] in a script written in Python to simulate the interaction, e.g. screen clicks and touches, of end users with the application. In fact, we were sending 5000 internal random events to the application with 2 milliseconds pause period between successive events. In addition, there are some behaviors in applications that only occur as a reaction to some external events, e.g. SMS_RECEIVED, in the OS. For the sake of completeness, we stimulate running application with some artificial external events such as incoming call, location updates, and SMS using capabilities of the OS emulator. Finally, after the execution of all events, which takes about 1 minute, in the virtual machine, we stopped the execution of application and revert the machine to the clean state by replacing it with the clean snapshot of the virtual machine that was taken before installing the application. Taking a snapshot and reverting to it afterwards are functionalities of VirtualBox and we used them by implementing a bash script.

6 EVALUATION

To evaluate our ServiceMonitor framework, we conducted a number of experiments. In what follows, we first describe the dataset, used in evaluation phase, in Section 6.1 and then afterwards ServiceMonitor framework is evaluated in Section 6.2. Furthermore, in Section 6.3, we discuss a number of observations made with respect to the different functionalities requested by the benign and malicious applications existing in the evaluation dataset.

6.1 Dataset

In order to evaluate the accuracy and performance of ServiceMonitor, we built a dataset of applications including benign and malicious samples. Our dataset of malicious applications was composed of 9560 samples, from 194 different families, obtained from AndroZoo [36], Drebin [6], and Malware Genome datasets [27]. The distribution of samples in each malicious family is illustrated in Fig. 5. Note that Drebin dataset was built for a static analysis approach. Therefore, a number of issues arise when employing dynamic analysis approaches on samples provided in this dataset. First, a number of samples in Drebin dataset could not be installed on an Android device. More specifically, we found that 429 samples have broken APK files. Second, we were unable to execute 1097 samples for analysis, either due to the fact that the application was dependent on the presence of another application, or the application executed as a background service. Considering the above issues, we excluded such applications from the dataset, hence our malicious set contained a total of 8034 samples.

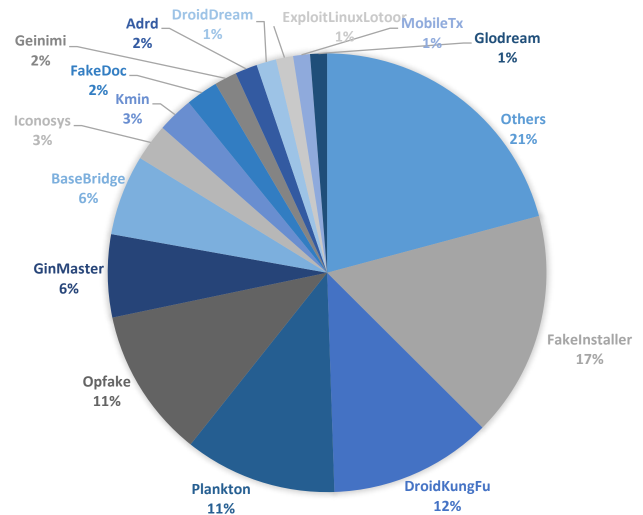

Figure 5: Distribution of samples in each malicious family in our initiated dataset.

Furthermore, we crawled the official Android store to collect 10370 samples from January 2019 to February 2019. In order to make sure that these samples were benign in nature, we submitted them to VirusTotal[37], which tests applications by fifty-four anti-malware engines. We found and eliminated 346 suspicious samples, therefore our benign dataset consisted of 10024 total number of samples.

6.2 Experimental Evaluation

In order to evaluate the effectiveness of ServiceMonitor, we employed four different classifiers: Random Forests [38], 1-Nearest Neighbor (1-NN) [39], 3-Nearest Neighbor (3-NN) [39], and Support Vector Machines (SVM) [40]. For this experiment, we trained classifiers with feature vectors obtained from Markov chain modeling procedure and also we used a k-fold cross-validation procedure with k=10 to overcome the over fitting problem. Furthermore, we should note that we extracted 51077 features for each application in the dataset. So, due to the machine learning problems with high dimensional data, it was difficult to have an efficient and accurate estimator in this case. As a remedy, in this step we used a well-suited feature selection method called Principal Component Analysis (PCA) [41]. In this statistical procedure, PCA ranks features by considering their variance in the feature space. In other words, we apply PCA to identify uncorrelated and most important features to reduce the dimension of feature space and improve the efficiency and accuracy of the detection system as well. After application of PCA, we reduced the dimension to 200 components/features with the highest contribution to the classification decision (i.e. features with maximum variance). Finally, we used the reduced-dimension feature vectors for learning the classifier and building the detection model.

The result of this experiment was encouraging. ServiceMonitor effectively classified the malicious and benign applications in the

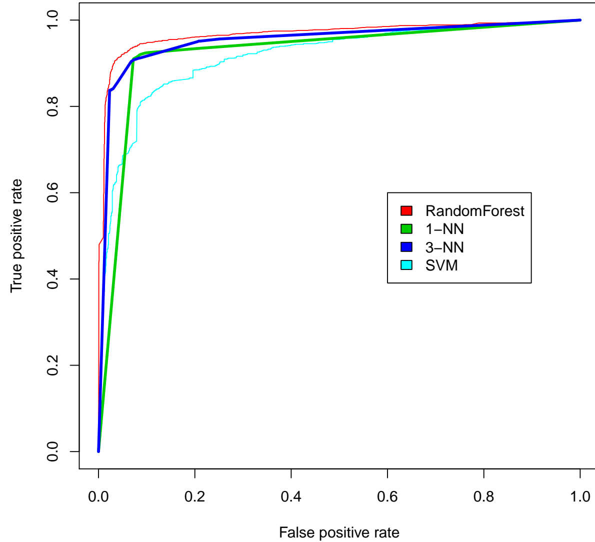

Figure 6: ROC curves obtained by evaluating the trained model, against the test dataset. Considering Random Forests algorithm, the area under ROC curve is 0.97.

introduced dataset with accuracy rate of 96.7%, false-negative rate of 4.4%, and false-positive rate of 2.1%. Fig. 6 depicts the ROC curves of this experiment to illustrate the true-positive rates against different false-positive values. Considering Random Forests algorithm, the calculated area under the ROC curve (i.e. AUC) is 0.97.

Even though ServiceMonitor is able to detect Android malware accurately, it is worth mentioning a number of issues which could limit its accuracy. Therefore, we describe the reasoning behind applications which were flagged as false-negative/-positive and some potential suggestions regarding how to overcome them in Section 7.

6.2.1 Runtime Measurement. With the aim of measuring the overhead of ServiceMonitor, we used PassMark PerformanceTest v2.0 [42] to benchmark the CPU and memory. In fact, PassMark conducts eight different tests to determine a device PassMark rating for CPU and memory. Since the benchmark results of PassMark test are returned as indexes, the higher value in this test means better performance. It is worth mentioning that we used a 2.5GHz quad-core phone with Android 6.0 as the evaluation platform.

As illustrated in Table 2, the overhead of ServiceMonitor on CPU and memory is acceptable. In fact, there is 0.8% and 2% performance impact on CPU and memory respectively. As shown, the highest overhead is on memory and this is mainly due to the IPC dissecting procedure that incurs overhead on Binder transactions.

6.2.2 Comparison with Related work. Among many proposals in the literature, we considered Patronus [13] as the state-of-the-art host-based approach, with which we could compare the proposed ServiceMonitor. However, as Patronus implemented system was

Table 2: Run time overhead of ServiceMonitor, measured by passmark benchmark.

Test	Baseline	ServiceMonitor	Overhead
CPU	13520	13410	0.8%
Memory	13860	13550	2%

Higher rating value means better performance.

not available, we were not able to evaluate Patronus based on the dataset employed in our evaluations. Hence and in order to conduct a valid and fair comparison, we employed a same dataset as used in Patronus, to evaluate ServiceMonitor.

With the same dataset, ServiceMonitor obtains an accuracy rate of 97.5% roughly 10% higher than Patronus; Furthermore, ServiceMonitor has false-positive rate of 0% against Patronus dataset, which is 1% lower than Patronus FPR. More importantly, Patronus is a policy-based solution, which is built on manually crafted malicious policies. Hence in contrast to ServiceMonitor, Patronus is not able to detect unknown malicious behaviors and zero-day malware. Also we should note that the overhead of ServiceMonitor is smaller than Patronus in the evaluated benchmark tests. Specifically, Patronus has 0.9% and 8% overhead on CPU and memory respectively in comparison with ServiceMonitor that has 0.8% and 2% overhead on CPU and memory respectively.

6.3 Observations

Given the large dataset employed in the evaluation of the proposed technique, we were able to obtain further insight into how applications, either benign or malicious, request access to different functionalities by means of system services.

1) Telephony Manager: As expected, requesting functions from telephony system services, such as functions related to retrieve phone number and unique device ID, i.e. IMEI, occurs more commonly in malware applications. More specifically, more than 67% of malware applications retrieve phone-related subscriber information, while there are only 9% of the benign applications with similar functionalities. In fact, telephony related methods such as *getDeviceId()*, *getLine1Number()*, *getSubscriberId()*, and *getIccSerialNumber()* are widely used in malicious applications in comparison with benign ones.

In addition, a great number of current malicious Android applications, such as malware samples belong to OpFake and Gemini families, subscribe to premium-rate services and send SMS messages to them with profitability objectives. Based on results, while 17% of malicious applications have telephony activities which cause financial charges to infected users, it was done by none of benign applications in our dataset.

2) Location Manager: Some malicious applications have access to GPS modules with the goal of collecting location data. For example, a malware family named AccuTrack is a family of applications that track down the GPS location of the device and turns it into a GPS tracker. In practice during the test, 12% of malware applications gained access to the location data, while only 7% of benign applications requested these services.

3) Package Manager: In our experiments, it is common that malware, such as some samples in DroidDream malware family, invoke *getInstalledPackages()* method to retrieve a list of all packages that are installed on the device to take an appropriate action. The result shows that about 16% of malicious applications requested such data during their test time while only 1% of benign applications requested it.

4) Activity Manager: getRunningAppProcesses() method from *ActivityManager* class was invoked in 3.7% of malicious applications to retrieve the list of running application processes; however, only 1% of the benign applications invoked this method. Applications can use this capability to check the presence of a running specific service, e.g. Anti-malware, in the device to take an action, e.g. killing the anti-malware process. *getMemoryInfo()* is another method used frequently by 5.7% of malicious applications to retrieve available memory space on the device, whereas only 0.7% of benign applications requested this method.

5) Service Manager: As a whole, results show that requesting system services in malware applications is more frequent than in benign applications, such that on average each malware requested 38 system services during its test time whereas less than 12 service requests occurred in each benign application. In fact, we considered occurrences of *getService()* method, from *ServiceManager* class, in Android applications to determine the average of service requests in them.

7 DISCUSSION

In this part, we aim to find out why some of applications in the dataset were misclassified. On a closer investigation, it is clear that most of false positives occur because the benign applications request system services in a similar way to the malicious ones. For example, social networking applications gain access to the most of the privacy-sensitive system resources (e.g. camera, location, gallery, SMS, etc) as well as having connections with remote servers. Although having unreal alarms is not user-friendly, it is clear that there is no security impact on the device taking false positives into account. One approach to decreasing the false positive rate is to combine other techniques such as signature-based verification with ServiceMonitor. However, further improvement of our solution in this direction is important future work but beyond the scope of this paper.

In addition, as noted in [43], detecting virtualization or emulation environments is one of the most popular methods employed by Android malware families, e.g. *Android.HeHe* [44] and *OBAD* [45], to evade analysis procedure and alter their behaviors accordingly. Therefore, due to the execution environment of ServiceMonitor training phase, which is based on a specific virtual machine, some malware could fingerprint the virtualized environment and avoid requests to system services and hence be classified as benign in our experiments (i.e. false negative). However, as an improvment to this limitation, Alzaylaee et al. [46] describe how an analysis environment can be configured to mimic a real device as much as possible and limit common methods used by malware for evading from analysis.

Furthermore, some malicious samples were unable to show their malicious behavior and classified as benign applications, i.e. false negative. For example, *C&C* servers of some malware were not available during the analysis time or the malicious logic maybe hidden and only executed, or triggered, under specific circumstances. Nevertheless, as an improving and complementary work, Wang et al. [47] proposed a system called Droid-AntiRM aims at detecting and taming such kind of anti-analysis techniques, i.e. triggering hidden malicious behavior under specific conditions.

As a learning-based method, ServiceMonitor might also be vulnerable to pollution and mimicry attacks [48]. In other words, malicious applications could randomly request system services and functionalities to change the original pattern of their requested functions from system services aim at confusing the detection system. However, Demontis et al. [49] proposed an adversary-aware machine-learning method which is able to improve a linear classifier against evasion attacks. Hence, in our future work, we will enhance our classifier with exploring the area of adversary-aware machine-learning algorithms.

8 CONCLUSIONS

In this paper, we proposed a system service use analysis technique that systematically extracts fine-grained behaviors of applications based on their accesses to system resources. Furthermore, we designed and implemented a host based system, called ServiceMonitor, that dynamically tracks execution behaviors of applications based on the proposed system service use analysis technique and models these behaviors in the form of Markov chains to classify applications into benign or malicious.

Our evaluation results show that ServiceMonitor is able to detect Android malware accurately and efficiently on mobile devices. Employing Random Forests classifier against 8034 malware and 10024 benign applications, ServiceMonitor were able to obtain the accuracy rate of 96.7% in distinguishing malicious applications from the benign ones.

ACKNOWLEDGMENTS

This research is supported by the research fund of KU Leuven and IMEC, a research institute founded by the Flemish government. The work of the third author has been partially supported by the EU H2020-SU-ICT-03-2018 Project No. 830929 CyberSec4Europe.

REFERENCES

[1] Gartner. (2018) Global mobile OS market share 2018. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems, Accessed: 2019-04-23.
[2] AV-Test.org. (2018) Security report 2017/18. https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2017-2018.pdf, Accessed: 2019-04-23.
[3] Google. Android-Google Play Protect. https://www.android.com/play-protect/, Accessed: 2019-04-23.
[4] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. J. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.
[5] V. P., A. Zemmari, and M. Conti, "A machine learning based approach to detect malicious android apps using discriminant system calls," *Future Generation Computer Systems*, vol. 94, pp. 333 – 350, 2019.
[6] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.
[7] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "Stormdroid: A streaminglized machine learning-based system for detecting android malware," in *Proceedings*

of the 11th ACM on Asia Conference on Computer and Communications Security. ACM, 2016, pp. 377–388.

[8] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "Droidsieve: Fast and accurate classification of obfuscated android malware," in *Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy*. ACM, 2017, pp. 309–320.

[9] K. Xu, Y. Li, and R. H. Deng, "Iccdetector: Icc-based malware detection on android," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1252–1264, June 2016.

[10] H.-J. Zhu, Z.-H. You, Z.-X. Zhu, W.-L. Shi, X. Chen, and L. Cheng, "Droiddet: effective and robust detection of android malware using static analysis along with rotation forest model," *Neurocomputing*, vol. 272, pp. 638–646, 2018.

[11] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "Madam: Effective and efficient behavior-based android malware detection and prevention," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1, pp. 83–97, Jan 2018.

[12] M. Sun, X. Li, J. C. S. Lui, R. T. B. Ma, and Z. Liang, "Monet: A user-oriented behavior-based malware variants detection system for android," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 5, pp. 1103–1112, 2016.

[13] M. Sun, M. Zheng, J. C. S. Lui, and X. Jiang, "Design and implementation of an android host-based intrusion prevention system," in *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*. ACM, 2014, pp. 226–235.

[14] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon: evaluating android anti-malware against transformation attacks," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. ACM, 2013, pp. 329–334.

[15] M. Y. Wong and D. Lie, "Tackling runtime-based obfuscation in android with TIRO," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018, pp. 1247–1262.

[16] V. M. Afonso, P. L. de Geus, A. Bianchi, Y. Fratantonio, C. Krügel, G. Vigna, A. Doupé, and M. Polino, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[17] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Krügel, and G. Vigna, "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[18] Y. Zhang, M. Yang, Z. Yang, G. Gu, P. Ning, and B. Zang, "Permission use analysis for vetting undesirable behaviors in android apps," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 11, pp. 1828–1842, 2014.

[19] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[20] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro, "DroidScribe: Classifying android malware based on runtime behavior," in *Proceedings of the IEEE Symposium Security and Privacy Workshops (SPW), Mobile Security Technologies (MoST)*, 2016, pp. 252–261.

[21] S. Bhandari, R. Panihar, S. Naval, V. Laxmi, A. Zemmari, and M. S. Gaur, "Sword: semantic aware android malware detector," *Journal of Information Security and Applications*, vol. 42, pp. 46 – 56, 2018.

[22] X. Xiao, Z. Wang, Q. Li, S. Xia, and Y. Jiang, "Back-propagation neural network on markov chains from system call sequences: a new approach for detecting android malware with system call sequences," *IET Information Security*, vol. 11, no. 1, pp. 8–15, 2016.

[23] F. Tong and Z. Yan, "A hybrid approach of mobile malware detection in android," *Journal of Parallel and Distributed Computing*, vol. 103, pp. 22 – 31, 2017.

[24] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Detecting android malware using sequences of system calls," in *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, ser. DeMobile 2015. ACM, 2015, pp. 13–20.

[25] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Comput. Surv.*, vol. 49, no. 4, pp. 76:1–76:41, Jan. 2017. [Online]. Available: http://doi.acm.org/10.1145/3017427

[26] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: A survey of issues, malware penetration, and defenses," *IEEE Communications Surveys Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

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

[28] M. Parkour. Mobile malware sharing site. http://contagiominidump.blogspot.com, Accessed: 2019-04-23.

[29] Google. (2019) Android developers. http://www.developer.android.com, Accessed: 2019-04-23.

[30] N. Artenstein and I. Revivo. (2014) Man in the binder: He who controls ipc, controls the droid. http://www.blackhat.com/docs/eu-14/materials/eu-14-Artenstein-Man-In-The-Binder-He-Who-Controls-IPC-Controls-The-Droid-wp.pdf, Accessed 2019-04-23.

[31] M. Salehi, F. Daryabar, and M. H. Tadayon, "Welcome to binder: A kernel level attack model for the binder in android operating system," in *Proceedings of the 8th International Symposium on Telecommunications (IST)*. IEEE, 2016, pp. 156–161.

[32] D.-H. You. Android platform based linux kernel rootkit. http://www.phrack.org/issues/68/6.html, Accessed: 2019-04-23.

[33] R Core Team. (2014) R: A language and environment for statistical computing. R Foundation for Statistical Computing. http://www.R-project.org, Accessed: 2019-04-23.

[34] Oracle vm virtualbox. http://www.qemu.org, Accessed: 2019-04-23.

[35] Monkey runner. http://developer.android.com/tools/help/monkey.html, Accessed: 2019-04-23.

[36] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 468–471.

[37] VirusTotal. (2019) Free online virus, malware and url scanner. https://www.virustotal.com, Accessed: 2019-04-23.

[38] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[39] E. Fix and J. L. Hodges Jr, "Discriminatory analysis-nonparametric discrimination: consistency properties," DTIC Document, Tech. Rep., 1951.

[40] M. A. Hearst, S. T. Dumais, E. Osman, J. Platt, and B. Scholkopf, "Support vector machines," *IEEE Intelligent Systems and their Applications*, vol. 13, no. 4, pp. 18–28, 1998.

[41] I. Jolliffe, *Principal component analysis*. Wiley Online Library.

[42] P. Software. (2019) Passmark performancetest. http://www.androidbenchmark.net, Accessed: 2019-04-23.

[43] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 2014, pp. 447–458.

[44] FireEye. Android.hehe. https://www.fireeye.com/blog/threat-research/2014/01/android-hehe-malware-now-disconnects-phone-calls.htm, Accessed: 2019-04-23.

[45] C. mobile mini dump. Obad. http://contagiominidump.blogspot.it/2013/06/backdoorandroidosobada.html, Accessed: 2019-04-23.

[46] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Dynalog: an automated dynamic analysis framework for characterizing android applications," in *Proceedings of the 2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*, 2016, pp. 1–8.

[47] X. Wang, S. Zhu, D. Zhou, and Y. Yang, "Droid-antirm: Taming control flow anti-analysis to support automated dynamic analysis of android malware," in *Proceedings of the 33th Annual Computer Security Applications Conference (ACSAC '17)*, 2017.

[48] S. Venkataraman, A. Blum, and D. Song, "Limits of learning-based signature generation with adversaries," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2008.

[49] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli, "Yes, machine learning can be more secure! a case study on android malware detection," *IEEE Transactions on Dependable and Secure Computing*, 2017.