Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2016, Article ID 8034967, 17 pages
http://dx.doi.org/10.1155/2016/8034967

Research Article

Detection and Visualization of Android Malware Behavior

Oscar Somarriba,[1,2] Urko Zurutuza,[1] Roberto Uribeetxeberria,[1] Laurent Delosières,[3] and Simin Nadjm-Tehrani[4]

[1] *Electronics and Computing Department, Mondragon University, 20500 Mondragon, Spain*
[2] *National University of Engineering (UNI), P.O. Box 5595, Managua, Nicaragua*
[3] *ExoClick SL, 08005 Barcelona, Spain*
[4] *Department of Computer and Information Science, Linköping University, 581 83 Linköping, Sweden*

Correspondence should be addressed to Oscar Somarriba; oscar.somarriba@gmail.com

Received 7 November 2015; Accepted 7 February 2016

Academic Editor: Aniket Mahanti

Malware analysts still need to manually inspect malware samples that are considered suspicious by heuristic rules. They dissect software pieces and look for malware evidence in the code. The increasing number of malicious applications targeting Android devices raises the demand for analyzing them to find where the malcode is triggered when user interacts with them. In this paper a framework to monitor and visualize Android applications' anomalous function calls is described. Our approach includes platform-independent application instrumentation, introducing hooks in order to trace restricted API functions used at runtime of the application. These function calls are collected at a central server where the application behavior filtering and a visualization take place. This can help Android malware analysts in visually inspecting what the application under study does, easily identifying such malicious functions.

1. Introduction

Collecting a large amount of data issued by applications for smartphones is essential for making statistics about the applications' usage or characterizing the applications. Characterizing applications might be useful for designing both an anomaly-detection system and/or a misuse detecting system, for instance.

Nowadays, smartphones running on an Android platform represent an overwhelming majority of smartphones [1]. However, Android platforms put restrictions on applications for security reasons. These restrictions prevent us from easily collecting traces without modifying the firmware or rooting the smartphone. Since modifying the firmware or rooting the smartphone may void the warranty of the smartphone, this method cannot be deployed on a large scale.

From the security point of view, the increase in the number of internet-connected mobile devices worldwide, along with a gradual adoption of LTE/4G, has drawn the attention of attackers seeking to exploit vulnerabilities and mobile infrastructures. Therefore, the malware targeting smartphones has grown exponentially. Android malware is one of the major security issues and fast growing threats facing the Internet in the mobile arena, today. Moreover, mobile users increasingly rely on unofficial repositories in order to freely install paid applications whose protection measures are at least dubious or unknown. Some of these Android applications have been uploaded to such repositories by malevolent communities that incorporate malicious code into them. This poses strong security and privacy issues both to users and operators. Thus, further work is needed to investigate threats that are expected due to further proliferation and connectivity of gadgets and applications for smart mobile devices.

This work focuses on monitoring Android applications' suspicious behavior at runtime and visualizing their malicious functions to understand the intention behind them. We propose a platform-independent behavior monitoring infrastructure composed of four elements: (i) an Android application that guides the user in selecting, instrumenting, and monitoring of the application to be examined, (ii) an embedded client that is inserted in each application to be

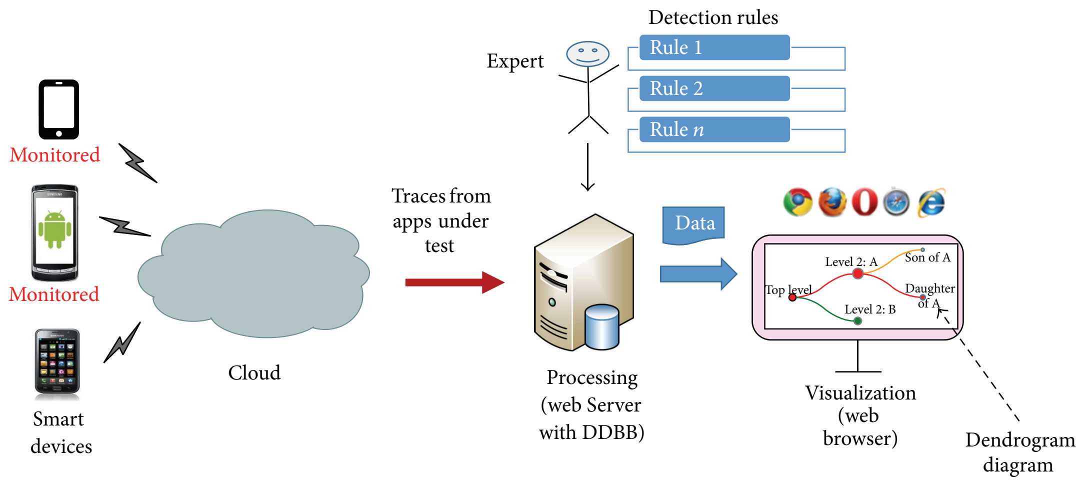


FIGURE 1: Overview of the monitoring system.

monitored, (iii) a cloud service that collects the application to be instrumented and also the traces related to the function calls, (iv) and finally a visualization component that generates behavior-related dendrograms out of the traces. A dendrogram [2] consists of many U-shaped nodes-lines that connect data of the Android application (e.g., the package name of the application, Java classes, and methods and functions invoked) in a hierarchical tree. As a matter of fact, we are interested in the functions and methods which are frequently seen in malicious code. Thus, malicious behavior could be highlighted in the dendrogram based on a predefined set of anomaly rules. An overview of the monitoring system is shown in Figure 1.

Monitoring an application at runtime is essential to understand how it interacts with the device, with key components such as the provided application programming interfaces (APIs). An API specifies how some software components (routines, protocols, and tools) should act when subject to invocations by other components. By tracing and analyzing these interactions, we are able to find out how the applications behave, handle sensitive data, and interact with the operating system. In short, Android offers a set of API functions for applications to access protected resources [3].

The remainder of the paper is organized as follows. Section 2 provides the notions behind the components used in the rest of the paper. Next, the related work is discussed in Section 3. Next we describe the monitoring and visualization architecture in Section 4, while we provide the details of the implementational issues of our system in Section 5. Later, in Section 6, we evaluate the proposed infrastructure and the obtained results by using 8 malware applications. Limitations and Conclusions are presented in Sections 7 and 8, respectively.

2. Background

Web Services extend the World Wide Web infrastructure to provide the means for software to connect to other software applications [4]. RESTFul Web Services are Web Services that use the principles of REpresentational State Transfer (REST) [5]. In other words, they expose resources to clients that can be accessed through the Hypertext Transfer Protocol (HTTP).

Regarding the Android operating system (OS), it is divided into four main layers: applications, application framework, middleware, and Linux kernel.

(i) Applications. The top layer of the architecture is where the applications are located. An Android application is composed of several components, amongst which we have Activities and Services. Activities provide a user interface (UI) of the application and are executed one at a time, while Services are used for background processing such as communication, for instance.

(ii) Application Framework. This layer is a suite of Services that provides the environment in which Android applications run and are managed. These programs provide higher-level Services to applications in the form of Java classes.

(iii) Middleware. This layer is composed of the Android runtime (RT) and C/C++ libraries. The Android RT is, at the same time, composed of the Dalvik Virtual Machine (DVM) (Android version 4.4. launches a new virtual machine called Android runtime (ART). ART has more advanced performance than DVM, among other things, by means of a number of new features such as the ahead-of-time (OTA) compilation, enhanced garbage collection, improved application debugging, and more accurate high-level profiling of the apps [6]) and a set of native (core) Android functions. The DVM is a key part of Android as it is the software where all applications run on Android devices. Each application that is executed on Android runs on a separate Linux process with an individual instance of the DVM, meaning that multiple instances of the DVM exist at the same time. This is managed by the Zygote process, which generates a fork of the parent DVM instance with the core libraries whenever it receives a request from the runtime process.

(iv) Linux Kernel. The bottom layer of the architecture is where the Linux kernel is located. This provides basic system functionality like process and memory management.

The kernel also handles a set of drivers for interfacing Android and interacting with the device hardware.

In standard Java environments, Java source code is compiled into Java bytecode, which is stored within .class format files. These files are later read by the Java Virtual Machine (JVM) at runtime. On Android, on the other hand, Java source code that has been compiled into .class files is converted to .dex files, frequently called Dalvik Executable, by the "dx" tool. In brief, the .dex file stores the Dalvik bytecode to be executed on the DVM.

Android applications are presented on an Android application package file (APK) .apk, the container of the application binary that contains the compiled .dex files and the resource files of the app. In this way, every Android application is packed using zip algorithm. An unpacked app has the following structure (several files and folders) [7]:

(i) an *AndroidManifest.xml* file: it contains the settings of the application (meta-data) such as the permissions required to run the application, the name of the application, definition of one or more components such as Activities, Services, Broadcasting Receivers, or Content Providers. Upon installing, this file is read by the *PackageManager*, which takes care of setting up and deploying the application on the Android platform.

(ii) a *res* folder: it contains the resources used by the applications. By resources, we mean the app icon, its strings available in several languages, images, UI layouts, menus, and so forth.

(iii) an *assets* folder: it stores noncompiled resources. This is a folder containing applications assets, which can be retrieved by *AssetManager*.

(iv) a *classes.dex* file: it stores the classes compiled in the dex file format to be executed on the DVM.

(v) a *META-INF* folder: this directory includes *MANIFEST.MF* which contains a cryptographic signature of the application developer certificate to validate the distribution.

The resulting .apk file is signed with a keystore to establish the identity of the author of the application. Besides, to build Android applications, a software developer kit (SDK) is usually available allowing access to APIs of the OS [8]. Additionally, two more components are described in order to clarify the background of this work: the Android-apktool [9] and the Smali/Backsmali tools. The Android-apktool is generally used to unpack and disassemble Android applications. It is also used to assemble and pack them. It is a tool set for reverse engineering third party Android apps that simplifies the process of assembling and disassembling Android binary .apk files into Smali .smali files and the application resources to their original form. It includes the Smali/Baksmali tools, which can decode resources (i.e., .dex files) to nearly original form of the source code and rebuild them after making some modifications. This enables all these assembling/disassembling operations to be performed automatically in an easy yet reliable way.

However, it is worth noting that the repackaged Android binary .apk files can only possess the same digital signature if the original keystore is used. Otherwise, the new application will have a completely different digital signature.

3. Related Work

Previous works have addressed the problem of understanding the Android application behavior in several ways. An example of inspection mechanisms for identification of malware applications for Android OS is presented by Karami et al. [10] where they developed a transparent instrumentation system for automating the user interactions to study different functionalities of an app. Additionally, they introduced runtime behavior analysis of an application using input/output (I/O) system calls gathered by the monitored application within the Linux kernel. Bugiel et al. [11] propose a security framework named *XManDroid* that extends the monitoring mechanism of Android, in order to detect and prevent application-level privilege escalation attacks at runtime based on a given policy. The principal disadvantage of this approach is that the modified framework of Android has to be ported for each of the devices and Android versions in which it is intended to be implemented. Unlike [10, 11], we profile only at the user level and therefore we do not need to root or to change the framework of Android smartphones if we would like to monitor the network traffic, for example.

Other authors have proposed different security techniques regarding permissions in Android applications. For instance, Au et al. [12] present a tool to extract the permission specification from Android OS source code. Unlike the other methods, the modules named Dr. Android and Mr. Hide that are part of a proposed and implemented app by Jeon et al. [13] do not intend to monitor any smart phones. They aim at refining the Android permissions by embedding a module inside each Android application. In other words, they can control the permissions via their module. We also embed a module inside each Android application but it is used to monitor the Android application instead.

In the work by Zhang et al. [3], they have proposed a system called *VetDroid* which can be described as a systematic analysis technique using an app's permission use. By using real-world malware, they identify the callsites where the app requests sensitive resources and how the obtained permission resources are subsequently utilized by the app. To do that, *VetDroid* intercepts all the calls to the Android API and synchronously monitors permission check information from Android permission enforcement system. In this way, it manages to reconstruct the malicious (permission use) behaviors of the malicious code and to generate a more accurate permission mapping than PScout [12]. Briefly this system [3] applies dynamic taint analysis to identify malware. Different from *VetDroid*, we do not need to root or jailbreak the phone nor do we conduct the permission-use approach for monitoring the smartphone.

Malware detection (MD) techniques for smart devices can be classified according to how the code is analyzed, namely, static analysis and dynamic analysis. In the former case, there is an attempt to identify malicious code by decompiling/disassembling the application and searching for suspicious strings or blocks of code; in the latter case the behavior of the application is analyzed using execution information. Examples of the two named categories are *Dendroid* [2] as an example of a static MD for Android OS devices and *Crowdroid* as a system that clusters system call frequency of applications to detect malware [14]. Also, hybrid approaches have been proposed in the literature for detection and mitigation of Android malware. For example, Patel and Buddhadev [15] combine Android applications analysis and machine learning (ML) to classify the applications using static and dynamic analysis techniques. Genetic algorithm based ML technique is used to generate a rules-based model of the system.

A thorough survey by Jiang and Zhou [16] charts the most common types of permission violations in a large data set of malware. Furthermore, in [17], a learning-based method is proposed for the detection of malware that analyzes applications automatically. This approach combines static analysis with an explicit feature map inspired by a linear-time graph kernel to represent Android applications based on their function call graphs. Also, Arp et al. [18] combine concepts from broad static analysis (gathering as many features of an application as possible) and machine learning. These features are embedded in a joint vector space, so typical patterns indicative of malware can be automatically identified in a lightweight app installed in the smart device. Shabtai et al. [19] presented a system for mobile malware detection that takes into account the analysis of deviations in application networks behavior (app's network traffic patterns). This approach tackles the challenge of the detection of an emerging type of malware with self-updating capabilities based on runtime malware detector (anomaly-detection system) and it is also standalone monitoring application for smart devices.

Considering that [17] and Arp et al. [18] utilize static methods, they suffer from the inherent limitations of static code analysis (e.g., obfuscation techniques, junk code to evade successful decompilation). In the first case, their malware detection is based upon the structural similarity of static call graphs that are processed over approximations, while our method relies upon real functions calls that can be filtered later on. In the case of Debrin, transformation attacks that are nondetectable by static analysis, as, for example, based on reflection and bytecode encryption, can hinder an accurate detection.

Although in [19] we have a detection system that continuously monitors app executions. There is a concern about efficiency of the detection algorithm used by this system. Unfortunately, in this case, they could not evaluate the Features Extractor and the aggregation processes' impact on the mobile phone resources, due to the fact that an extended list of features was taken into account. To further enhance the system's performance, it is necessary to retain only the most effective features in such a way that the runtime malware detector system yields relatively low overhead on the mobile phone resources.

Our proposed infrastructure is related to the approaches mentioned above and employs similar features for identifying malicious applications, such as permissions, network addresses, API calls, and function call graphs. However, it differs in three central aspects from previous work. First, we have a runtime malware detection (dynamic analysis) but abstain from crafting detection in protected environment as the dynamic inspections done by *VetDroid*. While this system provides detailed information about the behavior of applications, they are technically too involved to be deployed on smartphones and detect malicious software directly. Second, our visual analysis system is based on accurate API call graphs, which enables us to inspect directly the app in an easy-to-follow manner in the cloud. Third, we are able to monitor not just the network traffic, but most of the restricted and suspicious API calls in Android. Our platform is more dynamic and simpler than other approaches mentioned above.

General overview of the state of security in mobile devices and approaches to deal with malware can be found in [20], and in the work by Suarez-Tangil et al. in [21], as well as in recent surveys by Faruki et al. in [7] and Sufatrio et al. [6]. Malware in smart devices still poses many challenges and, in different occasions, a tool for monitoring applications at a large scale might be required. Given the different versions of Android OS, and with a rising number of device firmwares, modifying each of the devices might become a nontrivial task. This is the scenario in which the proposed infrastructure in this paper best fits. The core contribution of this work is the development of a monitoring and instrumentation system that allows a visual analysis of the behavior of Android applications for any device on which an instrumented application can run. In particular, our work results in a set of dendrograms that visually render existing API calls invoked by Android malware application, by using dynamic inspection during a given time interval, and visually highlighting the suspicious ones. Consequently, we aim to fill the void of visual security tools which are easy to follow for Android environments in the technical literature.

4. Platform Architecture

When Android applications are executed, they call a set of functions that are either defined by the developer of the application or are part of the Android API. Our approach is based on monitoring a desired subset of the functions (i.e., hooked functions) called by the application and then uploading information related to their usage to a remote server. The hooked function traces are then represented in a graph structure, and a set of rules are applied to color the graphs in order to visualize functions that match known malicious behavior.

For this, we use four components: the *Embedded client* and the *Sink* on the smartphone side, and the *Web Service* and the *Visualization component* on the remote server side.

A work flow depicting the main elements of the involved system is shown in Figure 2. In Stage 1, the application under study and a set of permissions are sent to the Web Service. Next, the main processing task of Stage 2, labeled as hooking process, is introduced. In this case, hooks or logging codes are inserted in the functions that require at least one of the permissions specified at the previous stage.

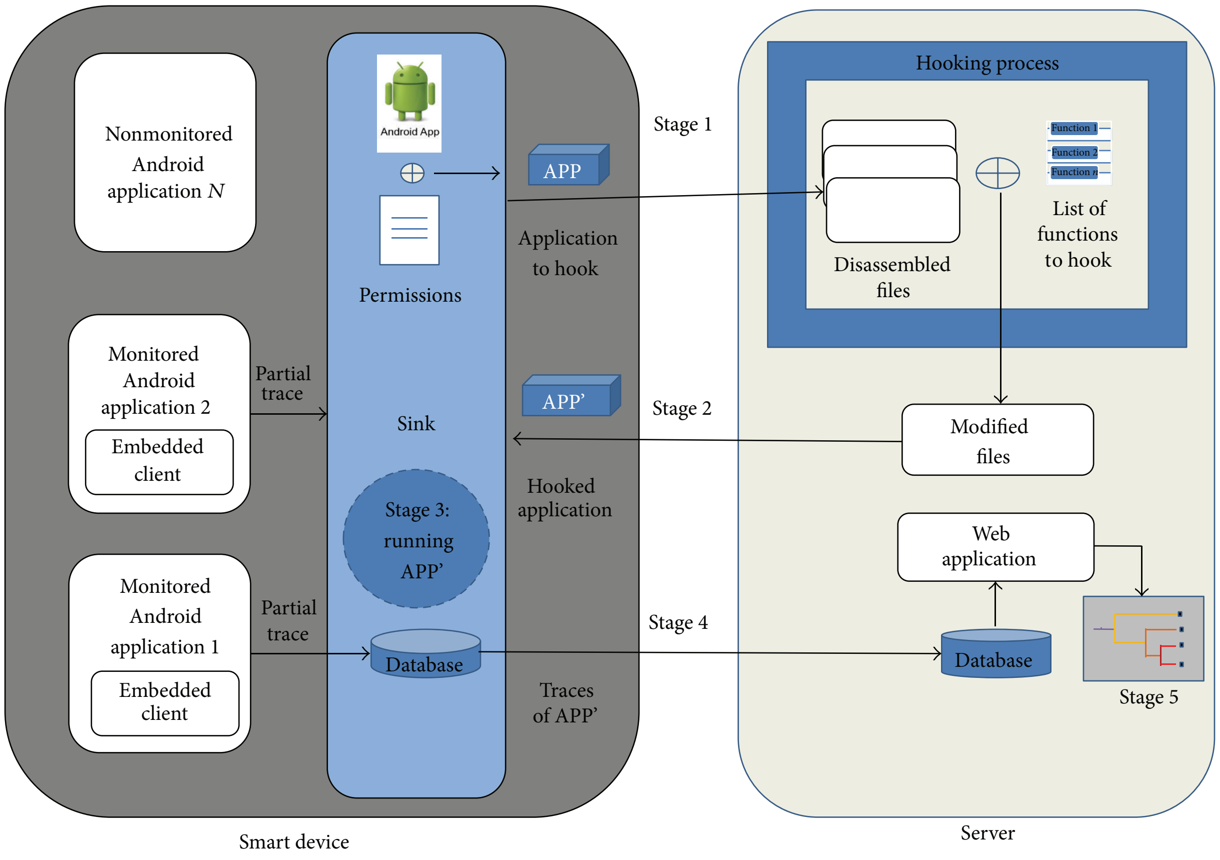


Figure 2: Schematics and logical stages of the system.

The new "augmented" application will be referred to as APP' from now on. Stages 3, 4, and 5 consist of running APP', saving the traces generated by APP' in the server's database, and showing the results as visualization graphs, respectively. The aforementioned infrastructure for platform-independent monitoring of Android applications is aimed to provide behavioral analysis without modifying the Android OS or root access to the smart device.

4.1. Embedded Client and Sink. The monitoring system consists of two elements: an embedded client that will be inserted into each application to be monitored and a Sink that will collect the hooked functions that have been called by the monitored applications. The embedded client simply consists of a communication module that uses the User Datagram Protocol (UDP) for forwarding the hooked functions to the Sink. Here, JavaScript Object Notation (JSON) is used when sending the data to the Sink, which allows sending dynamic data structures. In order to know the origin of a hooked function that has been received by the Sink, the corresponding monitored application adds its application hash, its package name, and its application name to the hooked function which we call a partial trace before sending it to the Sink.

The partial traces are built by the prologue functions (i.e., hook functions) that are placed just before their hooked functions and which modify the control flow of the monitored applications in order to build the partial traces corresponding with their hooked functions and passing the partial traces as parameter to the embedded client. Only the partial traces are built by the monitored application so that we add little extra overhead to the monitored application. The insertion of the embedded client and of the prologue functions in the Android application that is to be monitored is explained in Section 4.3.

The embedded client is written using the Smali syntax and is included on each of the monitored applications at the Web Service, at the same time that the functions hooks are inserted, before the application is packed back into an Android binary .apk file.

The Sink, on the other hand, is implemented as an Android application for portability both as a service and an activity whose service is started at the boot time. It is responsible for receiving the partial traces issued from all the monitored applications clients via a UDP socket, augmenting the partial traces to get a trace (i.e., adding a timestamp and the hash of the ID of the phone), storing them, and sending them over the network to the Web Service. As for the activity, it is responsible for managing the monitored applications via a UI, sending the applications to hook to the Web Service, and downloading the hooked applications from the Web Service. By hooked applications, we mean the applications in which hooks have been inserted. Once an application has been hooked then we can monitor it.

Before storing the traces in a local database, the Sink first stores them in a circular buffer which can contain up to 500 traces. The traces are flushed to the local database when any of the following conditions are met: (i) when the buffer is half full, (ii) when the Sink service is shutting down, or (iii) upon an activated timeout expiring. This bulk flushing enables the Sink to store the traces more efficiently. Unfortunately, if the service is stopped by force, we lose the traces that are present in the circular buffer. Once the traces are persisted in the local database, the timeout is rescheduled. Every hour, the Sink application tries to send the traces that remain in the local database out to the Web Service. A trace is removed locally upon receiving an acknowledgment from the Web Service. An acknowledgment is issued when the Web Service has been able to record the trace in a SQL database with success. If the client cannot connect to the Web Service, it will try again at the next round.

When a user wants to monitor an application, a message with the package name as payload is sent to the Sink service which keeps track of all the applications to monitor in a list. When a user wants to stop monitoring a given application, a message is sent to the Sink service which removes it from its list of applications to monitor.

4.2. The Web Service. This server provides the following services to *Sink*: upload applications, download the modified applications, and send the traces. Now the key part of the whole system, where the logic of the method presented lies, is the tool that implements the application, a process known as "hooking." In the following, we explain it. The Web Service, implemented as a Servlet on a Tomcat web application server, is a RESTful Web Service which exposes services to clients (e.g., Android smartphone) via resources. The Web Service exposes three resources which are three code pages enabling the Sink to upload an application to hook, download a hooked application, and send traces. The hooking process is explained in more detail in Section 4.3.1.

The file upload service allows the Sink to send the target application to monitor and triggers the command to insert all the required hooks and the embedded client to the application. Also, it is in charge of storing the submitted Android binary .apk file on the server and receiving a list of permissions. This set of permissions will limit the amount of hooks to monitor, hooking only the API function calls linked to these permissions. Conversely, the file download service allows the Sink to download the previously sent application, which is now prepared to be monitored. A ticket system is utilized in order to keep tracking of the current application under monitoring. The trace upstream service allows the Sink to upload the traces stored on the device to the server database and remove the traces from the devices local SQLite database. Upon receiving traces, the Web Service records them in a SQL database and sends an acknowledgment back to the Sink. In case of failure in the server side or in the communication channel, the trace is kept locally in the SQLite database until the trace is stored in the server and an acknowledgment is received by the Sink. In both cases, it might occur that the trace has just been inserted in the SQL database and no answer is sent back. Then the Sink would send again the same trace and we would get a duplication of traces. However, the mechanism of primary key implemented in the SQL database prevents the duplication of traces. A primary key is composed of one or more data attributes whose combination of values must be unique for each data entry in the database. When two traces contain the same primary key, only one trace is inserted while the insertion of the other one throws an exception. When such an exception is thrown, the Web Service sends back an acknowledgment to the Sink so as to avoid the Sink resending the same trace (i.e., forcing the Sink to remove from its local database the trace that has already been received by the Web Service).

4.3. Instrumenting an Application. In this section, we first describe the process of inserting hooks into an Android application and then we show an example of a hook implementation. A tutorial on instrumentation of Android applications is presented by Arzt et al. in [22].

However, before proceeding with the insertion of instrumentation code to the decompiled APK below, we would like to clarify the effect of disassembling the uploaded applications, that is, the differences between the original code and code generated after instrumentation. Briefly, the disassembling of the uploaded application is performed by using the Smali/Baksmali tool which is assembler/disassembler, respectively, for the dex-format (https://source.android.com/devices/tech/dalvik/dex-format.html). This is the format used by Dalvik, one of the Android's JVM implementations. Thus, the disassembling is able to recover an assembler-like representation of the Java original code. This representation is not the original Java source code (Baksmali is a disassembler, not a decompiler after all). However, Baksmali creates both an exact replica of the original binary code behavior and high-level enough to be able to manipulate it in an easy way. This is why we can add additional instructions to instrument the original code for our purposes and then reassemble it back to a dex file that can be executed by Android's JVM. On the other hand, as discussed in [22], instrumentation of applications outperforms static analysis approaches, as instrumentation code runs as part of the target app, having full access to the runtime state. So, this explains the rationale behind introducing hooks in order to trace core sensitive or restricted API functions used at runtime of the apps. In other words, the Smali code reveals the main restricted APIs utilized by the apps under test, even in the presence of source code obfuscation. We can therefore resort to monitoring these restricted APIs and keep tracking of those Android suspicious programs' behavior.

4.3.1. Hooks Insertion. The hooking process is done in 6 steps: (i) receiving the application to hook from the smartphone, (ii) unpacking the application and disassembling its Dalvik byte code via the Android-apktool, (iii) modifying the application files, (iv) assembling Dalvik byte code and packing the hooked application via the Android-apktool, (v) signing the

```
.class public Lcom/mainactivity/MainActivity;
...
invoke-static/range {v2 ··· v6}, log_sendTextMessage(···)
invoke-virtual/range {v1 ··· v6}, sendTextMessage(···)
...
```

Listing 1: Main activity class.

```
.class public Lorg/test/MonitorLog;
...
.method public static log_sendTextMessage(···)
...
const-string v0, “packageName: com.testprivacy,...”
invoke-static {v0}, sendLog(Ljava/lang/String;)
return-void
.end method

.method public static sendLog(Ljava/lang/String;)
.locals 3
.parameter payload
move-object v0, p0
...
new-instance v1, Ljava/lang/Thread;
new-instance v2, Lorg/test/EmbeddedClient;
invoke-direct {v2, v0}, init(Ljava/lang/String;)
invoke-direct {v1, v2}, init(Ljava/lang/Runnable;)
invoke-virtual {v1}, start()
return-void
.end method
```

Listing 2: Monitor log class.

hooked application, and (vi) sending the hooked application upon request of the smartphone.

Step (iii) can be subdivided into several substeps:

(1) adding the Internet permission in the *AndroidManifest* to enable the embedded client inserted in the application to hook to communicate with the Sink via UDP sockets,

(2) parsing the code files and adding invocation instructions to the prologue functions before their corresponding hooked functions: when the monitored application is running, before calling the hooked function, its corresponding prologue function will be called and will build its corresponding partial trace. The list of desired functions to hook is provided by the administrator of the Web Service. For instance, if the administrator is interested in knowing the applications usage, it will hook the functions that are called by the application when starting and when closing,

(3) adding a class that defines the prologue functions: it is worth noting that there will be as many prologue functions as functions to hook. Each prologue function builds its partial trace. Since we do not log the arguments of the hooked functions, the partial traces that are issued by the same monitored application will only differ by the name of the hooked function. It is also worth noting that the prologue functions are generated automatically.

Since every Android application must be signed by a certificate for being installed on the Android platform, we use the same certificate to check if the hooked application comes from our Web Service. For this, the certificate used in the Web Service has been embedded in the Sink application. This prevents attackers from injecting malicious applications by using a man-in-the-middle attack between the smartphone and the Web Service.

4.3.2. Hook Example. Consider a case where the function *sendTextMessage*, used to send short messages (SMS) on the Android platform, is to be logged in a monitored application. This function is called in the main activity class of the application corresponding to the code Listing 1. As for the class shown in Listing 2, it defines the prologue functions and the function responsible for passing the partial traces, built by the prologue functions, to the embedded client. For space reasons, we will not show the embedded client.

In the main activity class corresponding to the class shown in Listing 1, the function *sendTextMessage* is called at line (4) with its prologue function *log_sendTextMessage* which has been placed just before at line (3). Since the hooked function may modify common registers used for storing the parameters of the hooked function and for returning objects, we have preferred placing the prologue functions before their hooked functions. The register *v1* is the object of the class *SmSManager* needed to call the hooked function. As for the registers *v2* to *v6*, they are used for storing the parameters of the hooked function. Since our prologue functions are declared as static, we can call them without instantiating their class 2, and therefore we do not need to use the register *v1*.

An example of the monitor log class is shown in Listing 2. The name of the class is declared at line (1). At lines (3) and (10), two functions are defined, namely, *log_sendTextMessage* and *sendLog*. The former function, prologue function of the hooked function *sendTextMessage*, defines a constant string object containing the partial trace at line (5) and puts it into the register *v0*. Then the function *sendLog* is called at line (6) with the partial trace as parameter. The latter function saves the partial trace contained in the parameter *p0* into the register *v0* at line (13). At lines (15) and (16), two new instances are created, respectively: a new thread and new instance of the class *EmbeddedClient*. Their instances are initialized, respectively, at lines (17) and (18). Finally, the thread is started at line (19) and the partial trace is sent to the Sink. It is worth noting that, in these two examples, we have omitted some elements of the code which are replaced by dots to facilitate the reading of the code.

4.4. Visualization. The visualization of anomalous behavior is the last component of the proposed architecture. In order to perform a visual analysis of the applications' behavior in a simplified way, a D3.js (or just D3 for Data-Driven Documents (JavaScript library available at http://d3.org/)) graph was used. D3 is an interactive and a browser-based data visualizations library to build from simple bar charts to complex infographics. In this case, it stores and deploys graph oriented data on a tree-like structure named dendrograms using conventional database tables. Generally speaking, a graph visualization is a representation of a set of nodes and the relationships between them shown by links (*vertices* and *edges*, resp.).

This way, we are able to represent each of the analyzed application's behaviors with a simple yet illustrative representation. In general, the graphs are drawn according to the schema depicted in Figure 3. The first left-hand (root) node, "Application," contains the package name of the application, which is unique to each of the existing applications. The second middle node (parent), "Class," represents the name of the Android component that has called the API call. The third node, "Function" (the right-hand or child node), represents the names of functions and methods invoked by the application. It is worth noting that each application can include several classes and each class can call various functions or methods.

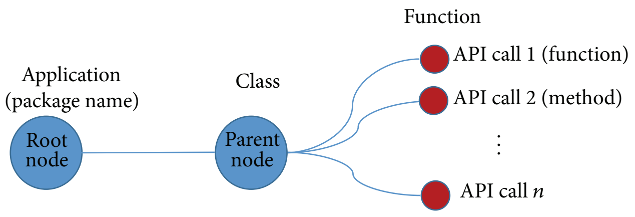


Figure 3: Schema used for the dendrograms.

In other words, function calls are located in the right-hand side of the dendrogram. For each node at this depth we are looking for known suspicious functions derived from a set of predefined rules as described below.

4.4.1. Rules "Generation". The rules aim to highlight restricted API calls, which allow access to sensitive data or resources of the smartphone and are frequently found in malware samples. These could be derived from the static analysis where the classes.dex file is converted to Smali format, as mentioned before, to get information considering functions and methods invoked by the application under test. On the other hand, it is well know that many types of malicious behaviors can be observed during runtime only. For this reason we utilize dynamic analysis; that is, Android applications are executed on the proposed infrastructure (see Figure 2) and interact with them. As a matter of fact, we are only interested in observing the Java based calls, which are mainly for runtime activities of the applications. This includes data accessed by the application, location of the user, data written to the files, phone calls, sending SMS/MMS, and data sent and received to or from the networks.

For the case that an application requires user interactions, we resort to do that manually so far. Alternatively, for this purpose one can use MonkeyRunner toolkit, which is available in Android SDK.

In [18, 23], authors list API functions calls that grant access to restricted data or sensible resources of the smartphone, which are very often seen in malicious code. We base our detection rules in those suspicious APIS calls. In particular, we use the following types of suspicious APIs:

(i) API calls for accessing sensitive data, for example, IMEI and USIMnumbeleakage, such as *getDeviceId()*, *getSimSerialNumber()*, *getImei()*, and *getSubscriberId()*,

(ii) API calls for communicating over the network, for example, *setWifiEnabled()* and *execHttpRequest()*,

(iii) API calls for sending and receiving SMS/MMS messages, such as *sendTextMessage()*, *SendBroadcast()*, and *sendDataMessage()*,

(iv) API calls for location leakage, such as *getLastKnownLocation()* and *getLatitude()*, *getLongitude()*, and *requestLocationUpdates()*,

(v) API function calls for execution of external or particular commands like *Runtime.exec()*, and *Ljava/lang/Runtime; -> exec()*,

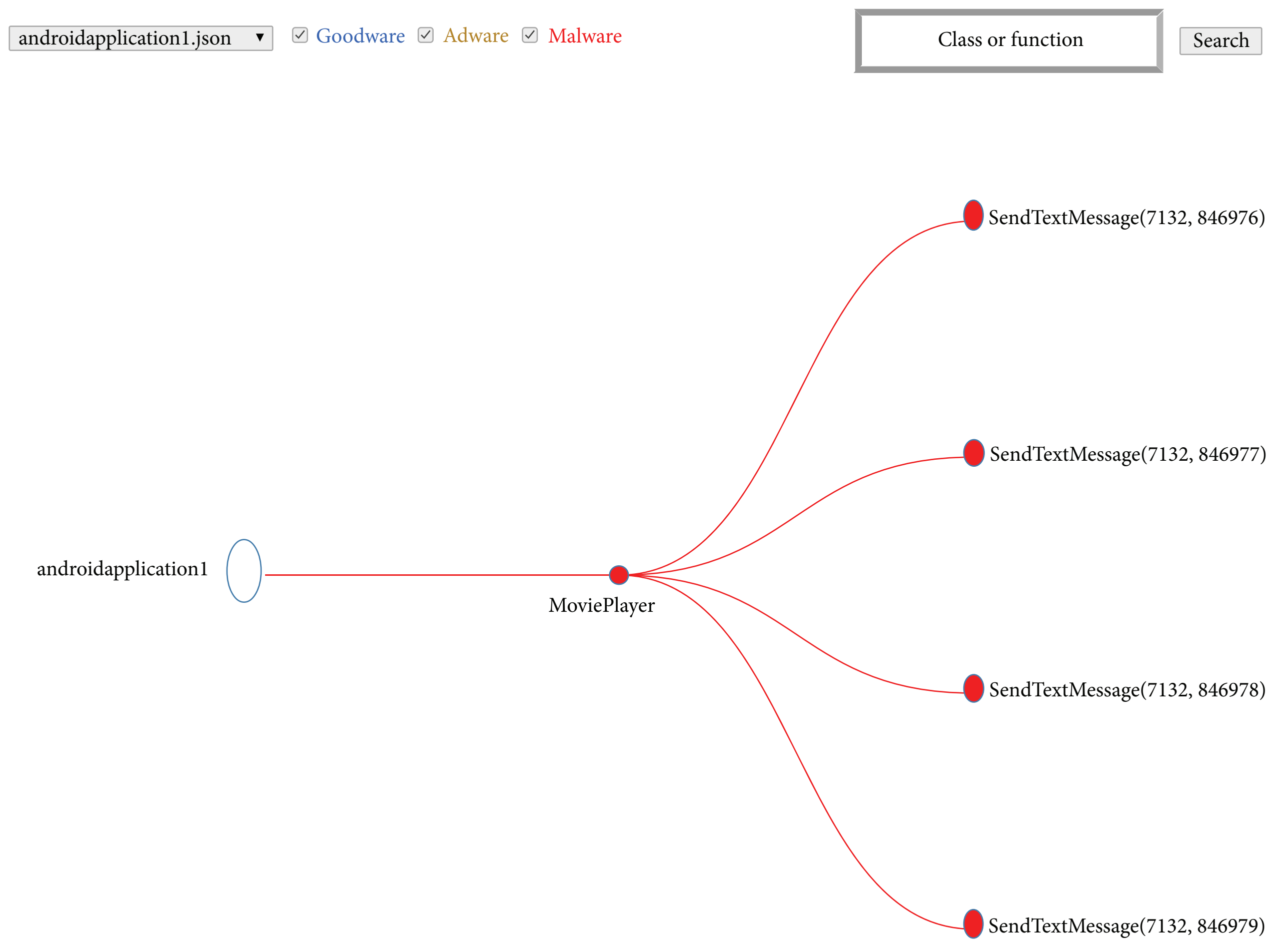

androidapplication1.json
Goodware
Adware
Malware
Class or function
Search
androidapplication1
MoviePlayer
SendTextMessage(7132, 846976)
SendTextMessage(7132, 846977)
SendTextMessage(7132, 846978)
SendTextMessage(7132, 846979)

Figure 4: The simplified dendrogram of the malware *FlakePlayer* has been generated using the D3. Note that at the upper left corner of the figure there is a combobox to select the monitored malware (here, for simplicity, we use a shortened version of package name of the app, i.e., *androidapplication1*). Besides, lining up to the right of the combobox, there are three activated checkboxes, labeled as Goodware in blue, Adware in orange, and Malware in red. Also, at the upper right corner of the figure, there is a *search button* that allows us to look for classes or functions. The complete package name of the malware *FakePlayer* is *org.me.androidapplication1.MoviePlayer*.

(vi) API calls frequently used for obfuscation and loading of code, such as *DexClassLoader.Loadclass()* and *Cipher.getInstance()*.

Here the rule module uses the above-mentioned API calls to classify the functions and methods invoked on the runtime of the applications into three classes, that is, Benign, Adware, or Malware. So in this way, we can generate IF-THEN rules (cf. rules-based expert systems). Next we show example rules that describe suspicious behavior. Some of the rules generated by us are similar or resemble the ones in [24], namely,

(1) a rule that shows that the examined app is not allowed to get the location of the smart device user:

IF Not (*ACCESS_FINE_LOCATION*) AND *CALL_getLastKnownLocation* THEN Malware,

(2) another rule which might detect that the application is trying to access sensitive data of the smartphone without permission:

IF Not (*READ_PHONE_STATE*) AND *CALL_getImei* THEN Malware.

Our approach selects from the database those functions that have been executed that match the suspicious functions described in the rules. Package name and class name of such function are colored accordingly to the "semaphoric" labeling described in Section 6.1.

To illustrate the basic idea we choose a malware sample, known as *FakePlayer*, in order to draw its graph. Thus, by means of running the filtering and visualization operations we end up with the graph of the malware, shown in Figure 4.

The system allows adding new rules in order to select and color more families of suspicious functions.

5. Testbed and Experimentation

Before introducing the reader into the results of using the monitoring and visualization platform, we need to explain the testbed. We first describe the experiment setup; then we follow the steps of running the client-side Sink.

5.1. Experiment Set Up. All the experiments have been realized on a Samsung Nexus S with Android Ice Cream

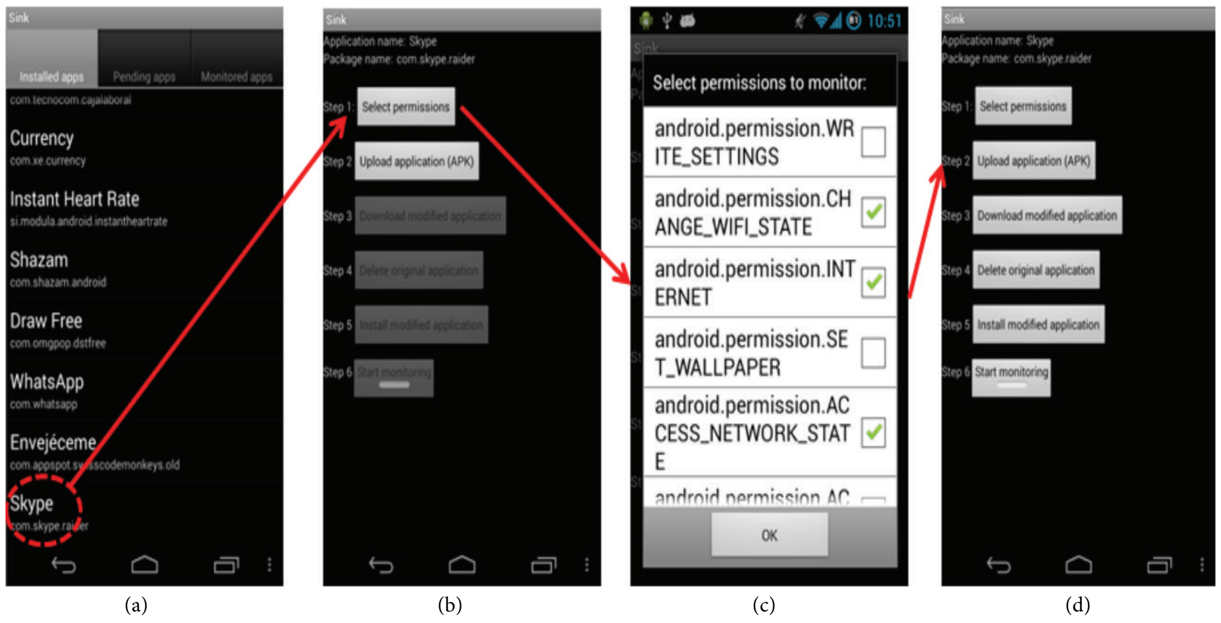

Sink
Installed apps
Pending apps
Monitored apps
com.tecnocom.cajalaboral
Currency
com.xe.currency
Instant Heart Rate
si.modula.android.instantheartrate
Shazam
com.shazam.android
Draw Free
com.omgpop.dstfree
WhatsApp
com.whatsapp
Envejéceme
com.appspot.swisscodemonkeys.old
Skype
com.skype.raider
Application name: Skype
Package name: com.skype.raider
Step 1 Select permissions
Step 2 Upload application (APK)
Step 3 Download modified application
Step 4 Delete original application
Step 5 Install modified application
Step 6 Start monitoring
10:51
Select permissions to monitor:
android.permission.WRITE_SETTINGS
android.permission.CHANGE_WIFI_STATE
android.permission.INTERNET
android.permission.SET_WALLPAPER
android.permission.ACCESS_NETWORK_STATE
android.permission.AC
OK
(a)
(b)
(c)
(d)

Figure 5: User interface of the Sink. (a) Choosing the application, (b) selecting the menu for permissions, (c) electing the permissions, and (d) steps of the monitoring process.

Sandwich (ICS). The Nexus S has a 1 GHz ARM Cortex A8 based CPU core with a PowerVR SGX 540 GPU, 512 MB of dedicated RAM, and 16 GB of NAND memory, partitioned as 1 GB internal storage and 15 GB USB storage.

We have explored different Android applications in order to evaluate the whole framework; some of these samples have been taken from the Android Malware Genome Project (The Android Malware Genome Project dataset is accessible at http://www.malgenomeproject.org/):

(i) *FakePlayer malware*,

(ii) *SMSReplicator malware*,

(iii) *iMatch malware*,

(iv) *DroidKungFu1 malware*,

(v) *DroidKungfu4 malware*,

(vi) *The spyware GoldDream in two flavors*,

(vii) *GGTracker malware*.

5.2. Client-Side Monitoring. The activities in Figure 5(a) display all the applications installed on the device that did not come preinstalled, from which the user selects a target application to monitor. Once an application is selected, the next step is to choose which permission or permissions the user wants to monitor. This can be observed in the third snapshot (white background) of Figure 5(c). Following the permissions clearance, the interface guides the user along several activities starting with the uploading of the selected application which is sent to the Web Service where the hooks are inserted. After this hooking process has finished, the modified application is downloaded from the Web Service. Afterwards, the original application is uninstalled and replaced by the modified application. Finally, a toggle allows starting and stoping monitoring of the application at any time by the user.

We focus on the functions of the Android API that require, at least, one permission. This allows the user to select from the Sink those permissions that are to be monitored at each application. This allows understanding how and when these applications use the restricted API functions. The PScout [12] tool was used to obtain the list of functions in the "API permission map." This way, the permission map obtained contains (Android 4.2 version API level 17) over thirty thousand unique function calls and around seventy-five different permissions. Besides, it is worth mentioning here that we refer to those associated with a sensitive API as well as sensitive data stored on device and privacy-sensitive built-in sensors (GPS, camera, etc.) as "restricted API functions." The first group is any function that might generate a "cost" for the user or the network. These APIs include [8], among others, Telephony, SMS/MMS, Network/Data, In-App Billing, and NFC (Near Field Communication) Access. Thus, by using the API map contained in the server's database, we are able to create a list of restricted ("suspicious") API functions.

The trace managing part is a service that runs in background with no interface and is in charge of collecting the traces sent from the individual embedded clients, located on each of the monitored applications. It adds a timestamp and the hash of the device ID and stores them on a common circular buffer. Finally, the traces are stored in bulk on a common local SQLite database and are periodically sent to the Web Service and deleted from the local database.

In summary, the required steps to successfully run an Android modified instrumented application are listed in Figure 5(d) and comprise the following.

Step 1 (select permissions). Set up and run the platform. Choose an application APP to be monitored on the device. Elect the permission list.

Step 2 (upload the application (APK)). Then, when this command is launched to upload the applications to the Web Service, the hooking process is triggered.

Step 3 (download modified application). This starts the downloading of the hooked application.

Step 4 (delete original application). This command starts the uninstallation process of the original application.

Step 5 (install modified application). This command starts the installation process of the modified application using Android's default application installation window.

Step 6 (start monitoring). Finally, a toggle is enabled and can be activated or disabled to start or stop monitoring that application as chosen by the user.

6. Results

To evaluate our framework, in this section we show the visualization results for several different applications to both benign and malicious. Then we proceed to evaluate the Sink application in terms of CPU utilization and ratio of partial traces received. Finally, we estimate the CPU utilization of a monitored application and its responsiveness.

6.1. Visual Analysis of the Traces. As mentioned before, a set of predefined rules allows us to identify the suspicious API functions and depending on its parameters (e.g., application attempts to send SMS to a short code that uses premium services) we assign colors to them. This enables us to quickly identify the functions and associate them with related items. On top of that, by applying the color classification of each node of the graph associated with a function in accordance with the color code (gray, orange, and red) explained below, it allows a "visual map" to be partially constructed. Furthermore, this graph is suitable to guide the analyst during the examination of a sample classified as dangerous because, for example, the red shading of nodes indicates malicious structures identified by the monitoring infrastructure.

In particular, to give a flavor to this analysis, the dendrogram of *FakePlayer* in Figure 4 provides the user with an indication of the security status of the malware. Different colors indicate the level of alarm associated with the currently analyzed application:

(i) Gray indicates that no malicious activity has been detected, as of yet.

(ii) Orange indicates that no malicious behavior has been detected in its graph, although some Adware may be presented.

(iii) Red indicates in its graph that a particular application has been diagnosed as anomalous, meaning that it contained one or more "dangerous functions" described in our blacklist. Moreover, it could imply the presence of suspicious API calls such as *sendTextMessage* with forbidden parameters, or the case of using restricted API calls for which the required permissions have not been requested (root exploit).

So, it is possible to conduct a visual analysis of the permissions and function calls invoked per application, where using some kind of "semaphoric labeling" allows us to identify easily the benign (in gray and orange colors) applications. For instance, in Figure 4 there is a presence of malware, and the nodes are painted in red.

The dendrogram shown for *FakePlayer* confirms its sneaky functionality by forwarding all the SMS sent to the device to the previously set phone number remaining unnoticed. For the sake of simplicity, we reduce the API function call *sendTextMessage*(phoneNo, null, SMS Content, null, null) to *sendTextMessage*(phoneNo, SMS Content). It uses the API functions to send four (see Figure 4) premium SMS messages with digit codes on it in a matter of milliseconds. Of course, sending a SMS message does not have to be malicious per se. However, for example, if this API utilizes numbers less that 9 digits in length, beginning with a "7" combined with SMS messages, this is considered a costly premium-rate service and a malware that sends SMS messages without the user's consent. The malware evaluated sends SMS messages that contain the following strings: 846976, 846977, 846978, and 846979. The message may be sent to a premium SMS short code number "7132," which may charge the user without his/her knowledge. This implies financial charges. Usually, when this malware is installed, malicious Broadcast Receiver is enrolled directly to broadcast messages from malicious server to the malware, so that user cannot understand whether specific messages are delivered or not. This is because the priority of malicious Broadcast Receiver is higher than SMS Broadcast Receiver. Once the malware is started, sending the function call *sendTextMessage* of SMS Manager API on the service layer, a message with premium number is sent which is shown in Figure 4.

6.2. Interactive Dendrograms. In general, it is needed to conduct the visual analysis from different perspectives. To do that we have developed an interactive graph visualization. So, we have four options or features in the D3 visualization of the application to monitor, namely, (a) selection of full features of the application (Goodware checkbox, Adware checkbox, and Malware checkbox), (b) the Goodware checkbox indicating that the app is assumed to be Goodware, (c) the Adware checkbox of the application, and (d) the Malware checkbox to look for malicious code. The analyst can choose to observe a particular Java class or function by typing the name of it inside the search box and clicking on the related search button.

Figures 6 and 7 illustrate a big picture of the whole behavioral performance of the malware *DroidKungFu1* whose package name is *com.nineiworks.wordsXGN*, and the malicious function calls are invoked. For the sake of simplicity, we shorten the package name of *DroidKungFu1* to *wordsXGN* in the dendrogram. As a matter of fact, we apply a similar

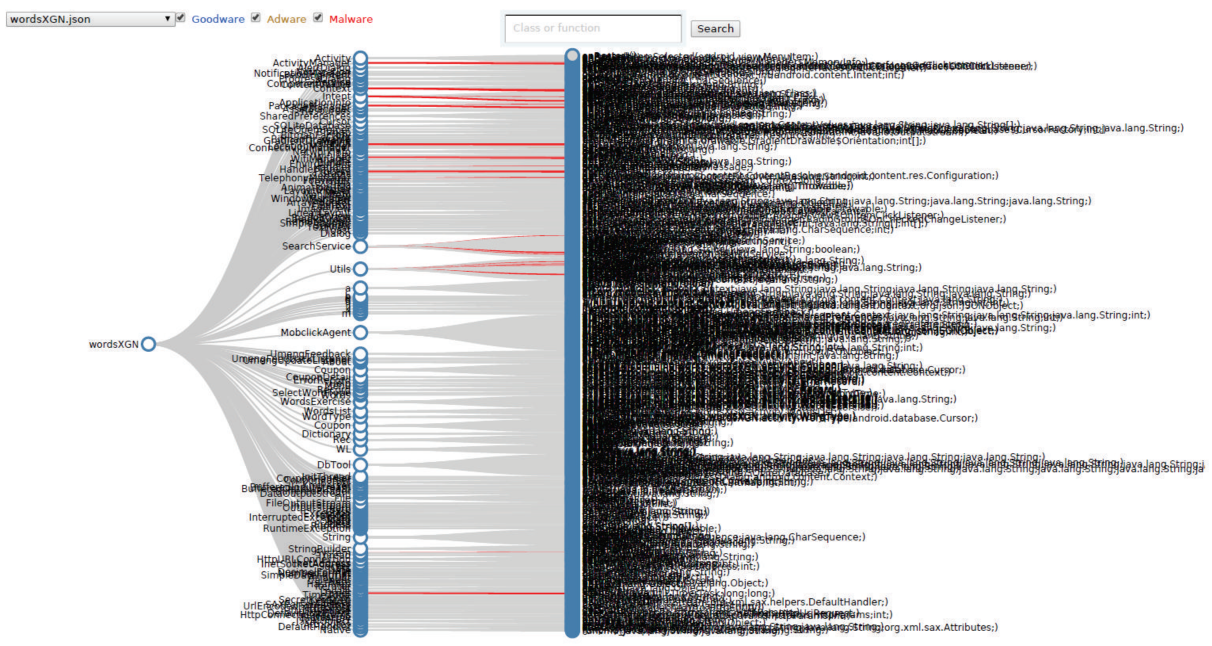
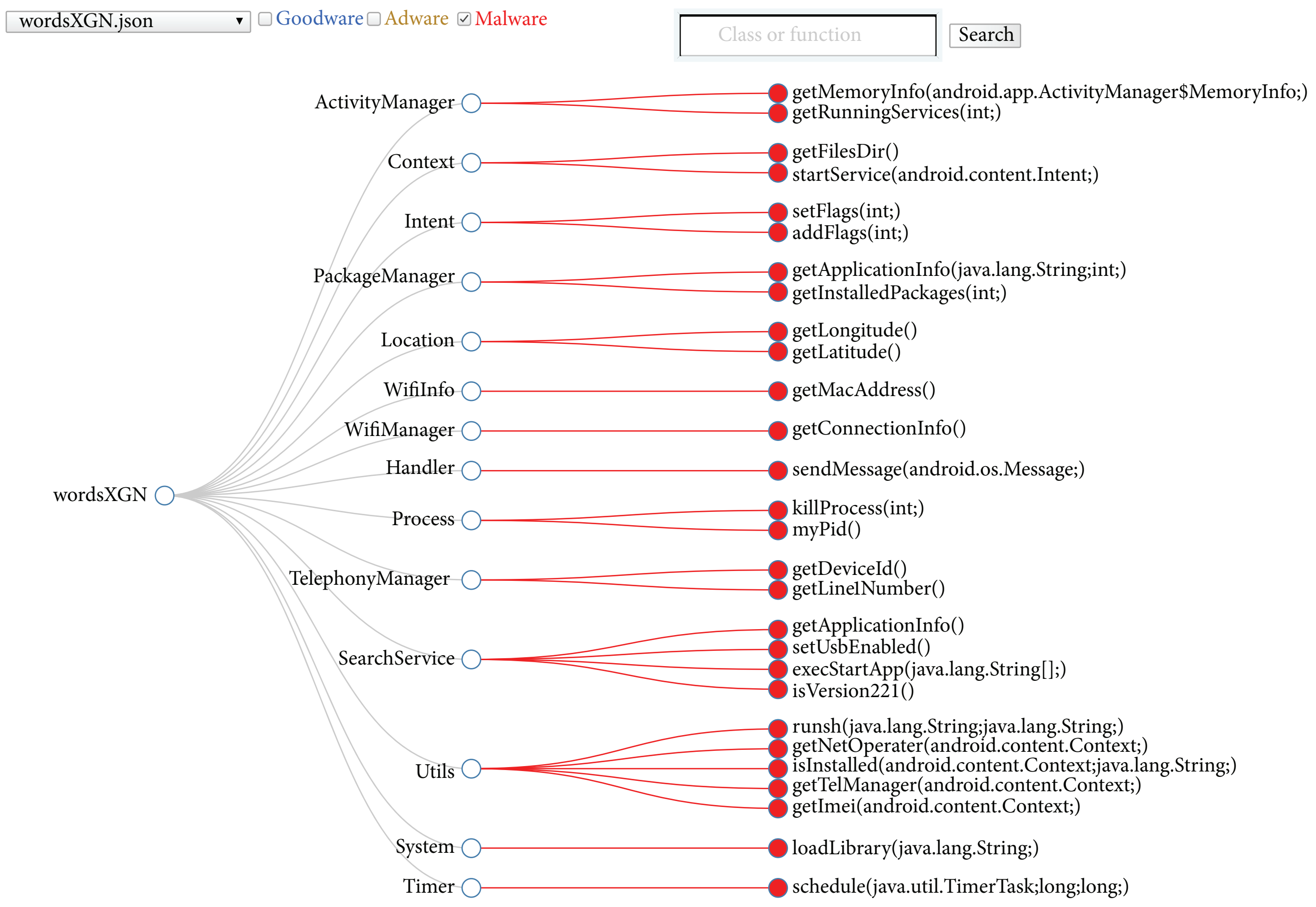


Figure 6: Visualization of the *DroidKungFu1* malware with full features chosen (i.e., all the checkboxes are activated).

Table 1: Malware family, detection rules, and suspicious functions.

Malware family	Detection rules	Suspicious functions
FakePlayer	IF (SEND_SMS) && (CALL_sendTextMessage() with preset numbers) THEN Malware	sendTextMessage(7132, null, 846976, null, null)
SMSReplicator	IF (SEND_SMS) && (CALL_sendTextMessage() with preset numbers) THEN Malware	sendTextMessage(1245, null, {From: 123456789 Hi how are you}, null, null)
iMatch	IF Not (ACCESS_FINE_LOCATION) && IF (SEND_SMS) THEN Malware	requestLocationUpdates(); sendTextMessage()
DroidKungFu1	[IF (INTERNET) && IF Not (ACCESS_FINE_LOCATION)] ‖ [IF (READ_PHONE_STATE) && IF (INTERNET)] THEN Malware	getLatitude(); getLongitude(); getDeviceid(); getLIne1Number(); getImei()
DroidKungFu4	IF (INTERNET) && IF (READ_PHONE_STATE) THEN Malware	getDeviceid(); getLIne1Number(); getSimSerial(); getImei();
GoldDream (Purman)	[IF (READ_PHONE_STATE) && IF Not (SEND_SMS)] ‖ [IF Not (READ_PHONE_STATE) && IF (INTERNET)] THEN Malware	getDeviceId(); getLIne1Number(); getSimSerial(); sendTextMessage(); getImei()
GoldDream (Dizz)	[IF (READ_PHONE_STATE) && IF Not (SEND_SMS)] ‖ [IF Not (ACCESS_FINE_LOCATION) && IF (INTERNET)] THEN Malware	getDeviceId(); getLIne1Number(); getSimSerial(); sendTextMessage(); requestLocationUpdates(); getImei()
GGTracker	[IF (READ_PHONE_STATE) && Not (SEND_SMS)] ‖ [IF Not (ACCESS_FINE_LOCATION) && IF (INTERNET)] THEN Malware	getDeviceId(); getLIne1Number(); getSimSerial(); sendTextMessage(); requestLocationUpdates(); getImei()

labeling policy to the other dendrograms. Moreover, we have the dendrograms for the *DroidKungFu4* in Figure 8. In particular, in the graph of Figure 8(a), we conduct the visual inspection by using full features (i.e., all the checkboxes active simultaneously) looking for red lines (presence of malware, if that is the case). Furthermore, in the graph of Figure 8(b), now we can focus our visual examination in the malicious functions carried out by the application. The visual analysis of the *DroidKungFu1* and *DroidKungFu4* includes encrypted root exploits, Command & Control (C & C) servers which in the case of *DroidKungFu1* are in plain text in a Java class file, and shadow payload (embedded app). In Table 1, we have some of the suspicious function calls utilized by the malware which pop up from the dendrograms. Regarding the IF-THEN rules, the allowed clauses or statements in our infrastructure are permissions and API functions calls. The fundamental operators are Conditional-AND which is denoted by &&, Conditional-OR which is denoted by ||,



wordsXGN.json ☐ Goodware ☐ Adware ☑ Malware

Class or function
Search
wordsXGN
ActivityManager
getMemoryInfo(android.app.ActivityManager$MemoryInfo;)
getRunningServices(int;)
Context
getFilesDir()
startService(android.content.Intent;)
Intent
setFlags(int;)
addFlags(int;)
PackageManager
getApplicationInfo(java.lang.String;int;)
getInstalledPackages(int;)
Location
getLongitude()
getLatitude()
WifiInfo
getMacAddress()
WifiManager
getConnectionInfo()
Handler
sendMessage(android.os.Message;)
Process
killProcess(int;)
myPid()
TelephonyManager
getDeviceId()
getLine1Number()
SearchService
getApplicationInfo()
setUsbEnabled()
execStartApp(java.lang.String[];)
isVersion221()
Utils
runsh(java.lang.String;java.lang.String;)
getNetOperater(android.content.Context;)
isInstalled(android.content.Context;java.lang.String;)
getTelManager(android.content.Context;)
getImei(android.content.Context;)
System
loadLibrary(java.lang.String;)
Timer
schedule(java.util.TimerTask;long;long;)

Figure 7: Visualization of the malicious API calls detected by our system for *DroidKungFu1*. Note the chosen options of the monitored malware in the dendrogram at the upper left side. First, we shorten version of the package name (*wordsXGN*) of the malware in the combobox. Next we have three checkboxes, namely, Goodware, Adware, and Malware. In this graph, only the red checkbox has been activated in order to conduct the visual analysis. The full package name of *DroidKungFu1* is *com.nineiworks.wordsXGN*.

and Not. For example, if the examined app does not have permission to send SMS messages in the *AndroidManifest* file and that app trys to send SMS messages with the location of the smartphone THEN that application may have malicious code. The rule generated for this case is shown below:

> IF Not (*SEND_SMS*) && (*ACCESS_FINE_LOCATION*) THEN Malware.

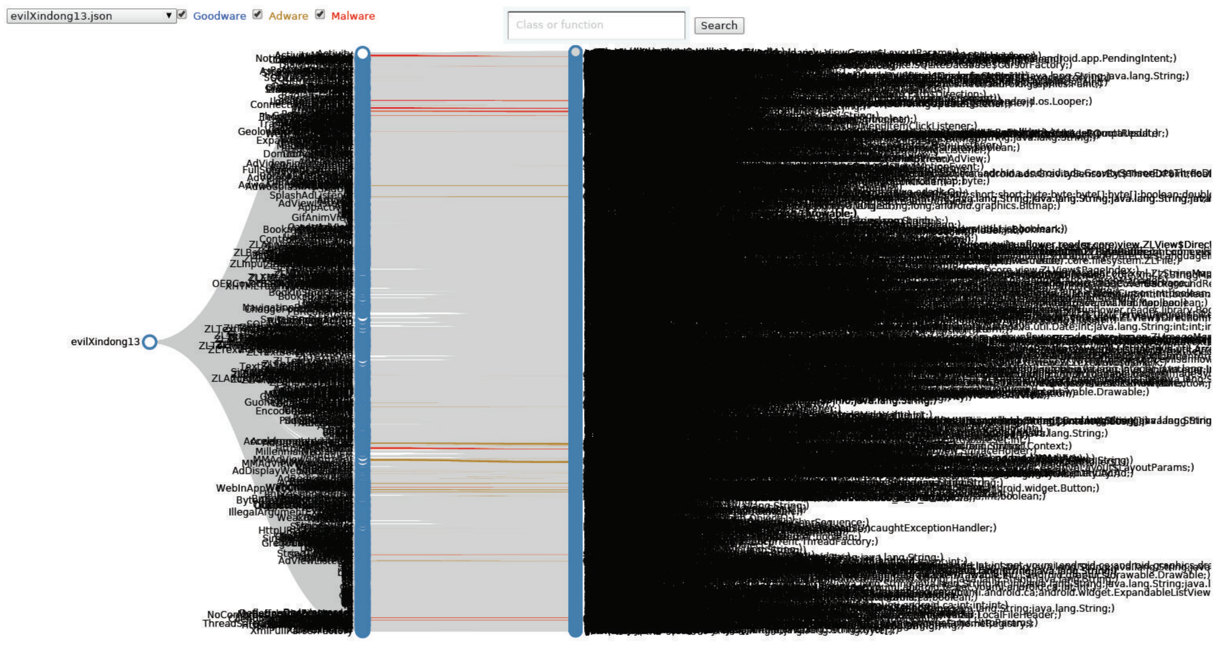
Here, malicious code and malware are interchangeable terms. The possible outcomes are Goodware or Malware. Nevertheless, the proposed infrastructure might be capable of evaluating a third option, Adware, in a few cases. In this paper we do not describe the IF-THEN rules for the third kind of outcome. In this work, we restrain the possible outcomes to the two mentioned options.

We have used 7 rules in our experimentation which are listed in Table 1 (note that rules 1 and 2 are the same). We have listed in Table 1 the most frequently used rules. They mainly cover cases of user information leakage.

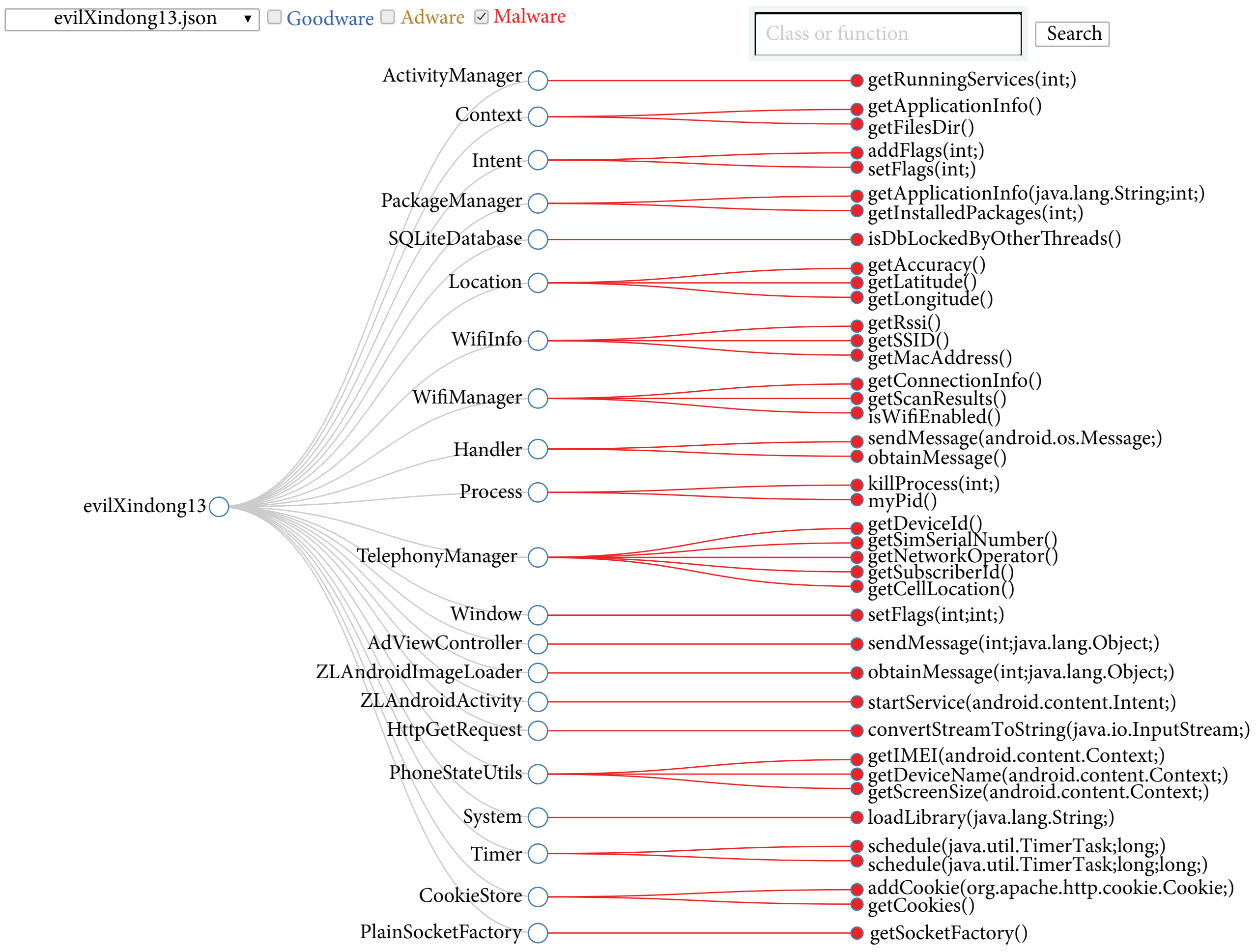
The most frequently used detection rules that we have utilized in our experimentation are listed in Table 1 (second column).

6.3. Client-Side CPU Use Analysis. We define the CPU utilization of a given application as the ratio between the time when the processor was in busy mode only for this given application at both the user and kernel levels and the time when the processor was either in busy or idle mode. The CPU times have been taken from the Linux kernel through the files "/proc/stat" and "/proc/pid/stat" where pid is the process id of the given application. We have chosen to sample the CPU utilization every second.

The CPU utilization of the Sink application has been measured in order to evaluate the cost of receiving the partial traces from the diverse monitored applications, processing them, and recording them in the SQLite database varying the time interval between two consecutive partial traces sent. We expect to see that the CPU utilization of the Sink increases as the time interval between two consecutive partial traces sent decreases. Indeed, since the Sink must process more partial traces, it needs more CPU resource. This is confirmed by the curve in Figure 9. The CPU utilization has a tendency towards 30% when the time interval between two consecutive partial traces received tends to 10 ms because the synthetic application takes almost 30% of the CPU for building and



(a) Dendrogram in full features (Goodware, Adware, and Malware) for *DroidKungFu4*



(b) Graph visualization of the detected malware in the case of *DroidKungFu4*

Figure 8: Dendrograms of the tested application. (a) Graph of the *DroidKungFu4* in full features, and (b) graph of the malicious functions invoked by *DroidKungFu4*. The full package name of DroidKungFu4 is com.evilsunflower.reader.evilXindong13.

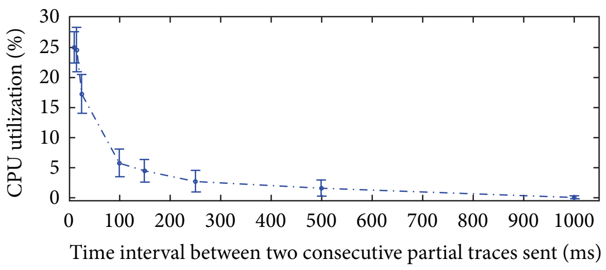


Figure 9: CPU utilization of the Sink.

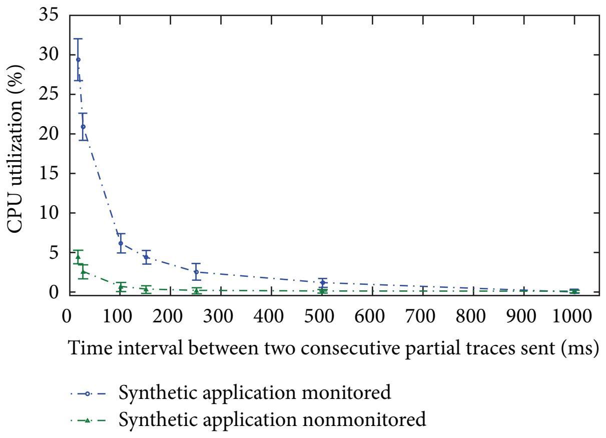


Figure 10: Difference of CPU utilization between an application monitored and nonmonitored.

sending partial traces, and the rest of applications utilize the rest of the CPU resource. When no monitored applications send partial traces to the Sink and the Sink is running in the background (i.e., its activity is not displayed on the screen), it consumes about 0% of the CPU.

The CPU utilization of a synthetic application has also been measured in order to evaluate the cost of building and sending the partial traces to the Sink while the time interval between two consecutive partial traces sent was varied. We expect to see a higher CPU utilization when the application is monitored. Indeed, since the synthetic application must build and send more partial traces, it needs more CPU resources. This is confirmed by Figure 10. We note that the increase of CPU utilization of the application can be up to 28% when it is monitored. The chart shows an increase in application CPU utilization to a level up to 38% which is justified when the monitoring is fine-grained at 10 ms. However, this high frequency is not likely to be needed in real applications.

6.4. Responsiveness. We define an application as responsive if its response time to an event is short enough. An event can be a button pushed by a user. In other words, the application is responsive if the user does not notice any latency while the application is running. In order to quantify the responsiveness and see the impact of the monitoring on the responsiveness of monitored applications, we have measured the time spent for executing the prologue function of the synthetic application. We have evaluated the responsiveness of the monitored application when the Sink was saturated by partial traces requests, that is, in its worst case. The measured response time was on average less than 1 ms. so, the user does not notice any differences when the application is monitored or not, even though the Sink application is saturated by partial traces. This is explained by the fact that UDP is connectionless and therefore sends the partial traces directly to the UDP socket of the Sink without waiting for any acknowledgments.

7. Limitations

So far we illustrated the possibilities of our visual analysis framework by analyzing 8 existing malicious applications. We successfully identified different types of malware accordingly to the malicious payload (e.g., privilege escalation, financial charges, and personal information stealing) of the app while using only dynamic inspection in order to obtain the outcomes. Even though the results are promising, they only represent a few of the massive malware attacking today's smart devices. Of course, the aim of this system is not to replace existing automated Android malware classification systems because the final decision is done by a security analyst.

Although, here, we propose a malware detector system based on runtime behavior, this does not have detection capabilities to monitor an application's execution in real time, so this platform cannot detect intrusions while running. It only enable detecting past attacks.

Also, one can figure out that malware authors could try avoiding detection, since they can gain knowledge whether their app has been tampered with or no. As a result, the actual attack might not be deployed, which may be considered a preventive technique. Moreover, it is possible for a malicious application to evade detection by dynamically loading and executing Dalvik bytecode at runtime.

One of the drawbacks of this work could be the manual interactions with the monitored application during runtime (over some time interval). Also, the classification needs a more general procedure to get the rule-based expert system. The natural next step is to automate these parts of the process. For example, in the literature there are several approaches that can be implemented in order to automatically generate more IF-THEN rules [15] or to resort to the MonkeyRunner kit available in Android SDK to simulate the user interactions. Of course, the outcomes of the 8-sample malware presented here are limited to longest time interval used in the study, which was 10 minutes. Extending this "playing" time with the app using tools for the automation of user's interactions could provide a more realistic graph and better pinpoint the attacks of the mobile malware.

Another limitation of this work is that it can only intercept Java level calls and not low level functions that can be stored as libraries in the applications. Thus, a malicious app can invoke native code through Java Native Interface (JNI), to deploy attacks to the Android ecosystem. Since our approach builds on monitoring devices that are not rooted, this approach is out of the scope of our research.

It is worth mentioning that our API hooking process does not consider the Intents. The current version of the infrastructure presented in this paper is not capable of monitoring the Intents sent by the application, as sending Intents does not require any kind of permission. Not being able to monitor Intents means that the infrastructure is not able to track if the monitored application starts another app for a short period of time to perform a given task, for instance, opening a web browser to display the end-user license agreement (EULA). Also, adding this feature would allow knowing how the target application communicates with the rest of the third party and system applications installed on the device.

Ultimately, this framework could be useful for final users interested in what apps are doing in their devices.

8. Conclusions

We provide a monitoring architecture aiming at identifying harmful Android applications without modifying the Android firmware. It provides a visualization graph named dendrograms where function calls corresponding to predefined malware behaviors are highlighted. Composed of four components, namely, the embedded client, the Sink, the Web Service, and the visualization, any Android application can be monitored without rooting the phone or changing its firmware.

The developed infrastructure is capable of monitoring simultaneously several applications on various devices and collecting all the traces in the same place. The tests performed in this work show that applications can be prepared to be monitored in a matter of minutes and that the modified applications behave as they were originally intended to, with minimal interference with the permissions used for. Furthermore, we have shown that the infrastructure can be used to detect malicious behaviors by applications, such as the monitored *FakePlayer*, *DroidKungFu1* and *DroidKungFu4*, and the *SMSReplicator* and many others taken from the dataset of the Android Malware Genome Project.

Evaluations of the Sink have revealed that our monitoring system is quite reactive, does not lose any partial traces, and has a very small impact on the performance of the monitored applications.

A major benefit of the approach is that the system is designed as platform-independent so that smart devices with different versions of Android OS can use it. Further improvements on the visualization quality and the user interface are possible, but the proof of concept implementation is demonstrated to be promising. For future work, we plan to extend the current work in order to develop a real-time malware detection infrastructure based on network traffic and on a large number of apps.

Competing Interests

The authors declare that they have no competing interests.

Acknowledgments

This research has been partially funded by Basque Governments Elkartek Program under Grant no. KK-2015/0000080. Also, the last author would like to acknowledge the support by the RICS Centre for Resilient Information and Control Systems (http://www.rics.se/) financed by the Swedish civil contingencies agency.

References

[1] Gartner, "Gartner says worldwide traditional PC, tablet, ultramobile and mobile phone shipments on pace to grow 7.6 percent in 2014," October 2015, http://www.gartner.com/newsroom/id/2645115.

[2] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, "Dendroid: a text mining approach to analyzing and classifying code structures in Android malware families," *Expert Systems with Applications*, vol. 41, no. 4, pp. 1104–1117, 2014.

[3] Y. Zhang, M. Yang, B. Xu et al., "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS '13)*, pp. 611–622, ACM, Berlin, Germany, November 2013.

[4] Microsoft, "Web services," 2014, http://msdn.microsoft.com/en-us/library/ms950421.aspx.

[5] R. T. Fielding, *Architectural styles and the design of network-based software architectures [Ph.D. thesis]*, University of California, Irvine, Calif, USA, 2000.

[6] Sufatrio, D. J. J. Tan, T.-W. Chua, and V. L. L. Thing, "Securing android: a survey, taxonomy, and challenges," *ACM Computing Surveys*, vol. 47, no. 4, article 58, 2015.

[7] P. Faruki, A. Bharmal, V. Laxmi et al., "Android security: a survey of issues, malware penetration, and defenses," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[8] Android.com, "Android system permissions," 2014, http://developer.android.com/guide/topics/security/permissions.html.

[9] R. Winsniewski, "Android-apktool: a tool for reverse engineering android apk files," 2012.

[10] M. Karami, M. Elsabagh, P. Najafiborazjani, and A. Stavrou, "Behavioral analysis of android applications using automated instrumentation," in *Proceedings of the 7th International Conference on Software Security and Reliability (SERE-C '13)*, pp. 182–187, Gaithersburg, Md, USA, June 2013.

[11] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, "Xmandroid: a new android evolution to mitigate privilege escalation attacks," Tech. Rep. TR-2011-04, Technische Universität Darmstadt, Darmstadt, Germany, 2011.

[12] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: analyzing the Android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, pp. 217–228, ACM, Raleigh, NC, USA, October 2012.

[13] J. Jeon, K. K. Micinski, J. A. Vaughan et al., "Dr. android and Mr. hide: fine-grained permissions in android applications," in *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '12)*, pp. 3–14, ACM, Raleigh, NC, USA, October 2012.

[14] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behaviorbased malware detection system for android," in *Proceedings of the 1st ACM Workshop on Security And Privacy in Smartphones and Mobile Devices*, pp. 15–26, ACM, 2011.

[15] K. Patel and B. Buddhadev, "Predictive rule discovery for network intrusion detection," in *Intelligent Distributed Computing*,

vol. 321 of *Advances in Intelligent Systems and Computing*, pp. 287–298, Springer, Basel, Switzerland, 2015.

[16] X. Jiang and Y. Zhou, *Android Malware*, Springer, New York, NY, USA, 2013.

[17] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," in *Proceedings of the ACM Workshop on Artificial Intelligence and Security*, pp. 45–54, ACM, 2013.

[18] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: effective and explainable detection of android malware in your pocket," in *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS '14)*, San Diego, Calif, USA, February 2014.

[19] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici, "Mobile malware detection through analysis of deviations in application network behavior," *Computers & Security*, vol. 43, pp. 1–18, 2014.

[20] M. La Polla, F. Martinelli, and D. Sgandurra, "A survey on security for mobile devices," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 1, pp. 446–471, 2013.

[21] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda, "Evolution, detection and analysis of malware for smart devices," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 2, pp. 961–987, 2014.

[22] S. Arzt, S. Rasthofer, and E. Bodden, "Instrumenting android and java applications as easy as abc," in *Runtime Verification*, pp. 364–381, Springer, Berlin, Germany, 2013.

[23] S.-H. Seo, A. Gupta, A. M. Sallam, E. Bertino, and K. Yim, "Detecting mobile malware threats to homeland security through static analysis," *Journal of Network and Computer Applications*, vol. 38, no. 1, pp. 43–53, 2014.

[24] K. Patel and B. Buddadev, "Detection and mitigation of android malware through hybrid approach," in *Security in Computing and Communications*, vol. 536 of *Communications in Computer and Information Science*, pp. 455–463, Springer, Basel, Switzerland, 2015.