Chapter 10

Malware Analysis and Detection on Android: The Big Challenge

Abraham Rodríguez-Mota,
Ponciano J. Escamilla-Ambrosio and
Moisés Salinas-Rosales

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/intechopen.69695

Abstract

The popularization of the use of mobile devices, such as smartphones and tablets, has accelerated in recent years, as these devices have experienced a reduction in cost together with an increase in functionality and services availability. In this context, due to its openness and free availability, Android operating system (OS) has become not only a major stakeholder in the market of mobile devices but has also become an attractive target for cybercriminals. In this chapter, we advocate to present some current trends and results in the Android malware analysis and detection research area. We start by briefly describing the Android's security model, followed by a discussion of the static and dynamic malware analysis techniques in order to provide a general view of the analysis and detection process to the reader. After that, a description of a particular set of software developments, which exemplify some of the discussed techniques, is presented accompanied by a set of practical results. Finally, we draw some conclusions about the future development of the Android malware analysis area. The main contribution of this chapter is a description of the realization of static and dynamic malware analysis techniques and principles that can be automated and mapped to software system tools in order to simplify analyses. Moreover, some details about the use of machine learning algorithms for malware classifications and the use of the hooking software techniques for dynamic analysis execution are provided.

Keywords: malware analysis, android, mobile devices, threat detection, cybersecurity

1. Introduction

Nowadays, mobile devices such as smartphones and tablets have become very popular, due to a reduction in their cost and an increase in their functionalities and services availability.

Moreover, the growing trend of implementing bring your own device (BYOD) policies in organizations has also contributed to the adoption of these technologies, not only for everyday communication activities but to support enterprise systems, industrial applications, and commercial transactions, which raise new security issues. In this scenario, operating systems have also played an important role in the adoption and proliferation of mobile devices and applications, giving also space for the appearance of malicious software (malware). This is the case for the Android OS, which, due to its openness and free availability, has become not only a major stakeholder in the market of mobile devices but has also become an attractive target for cybercriminals.

Google, the Open Handset Alliance manufacturers, and the Android developers' community have made many efforts in order to improve Android's security. However, the emergence and evolution of new security threats continue being an important issue. Therefore, in this chapter, we advocate to present some current trends and results in the Android malware analysis and detection research area. We start by briefly describing the Android's security model, followed by a discussion of some static and dynamic malware analysis techniques in order to provide a general view to the analysis and detection processes to the reader. After that, a description of a particular set of software developments, which exemplify some of the discussed techniques, is presented accompanied by a set of practical results. Finally, a set of conclusions about the future development of the ideas explored in this chapter are drawn.

2. Android security architecture

In a general sense, Android is not only an OS but a platform of three main building blocks: device hardware, Android OS, and the application runtime, see **Figure 1**.

First of all, the Android device hardware block refers to the wide range of hardware configurations where Android can be run, including smartphones, tablets, watches, automobiles, smart TVs, OTT gaming boxes, and set top boxes. Android is processor-agnostic, but it does take advantage of some hardware-specific security capabilities such as ARM eXecute-Never. Secondly, the Android OS building block refers to the Android OS itself, which is built on top of the Linux kernel, thus all device resources are accessed through the operating system. Thirdly, the Android application runtime block refers to the managed runtime used by applications and some system services on Android [2]. In this case, it must be taken into account that applications are written in the Java language and run in the Android runtime (ART). However, many applications, including core Android services and applications, are native applications or included native libraries. Both ART and native applications run with the same security environment, contained with the applications sandbox. Thus, applications get a dedicated part of the file system in which they can write private data, including database and raw files [1].

In this context, in terms of security, Android incorporates industry-leading security features and works with developers and device implementers to keep the Android platform and ecosystem safe. It was designed with multi-layered security that is flexible enough to support an

http://dx.doi.org/10.5772/intechopen.69695

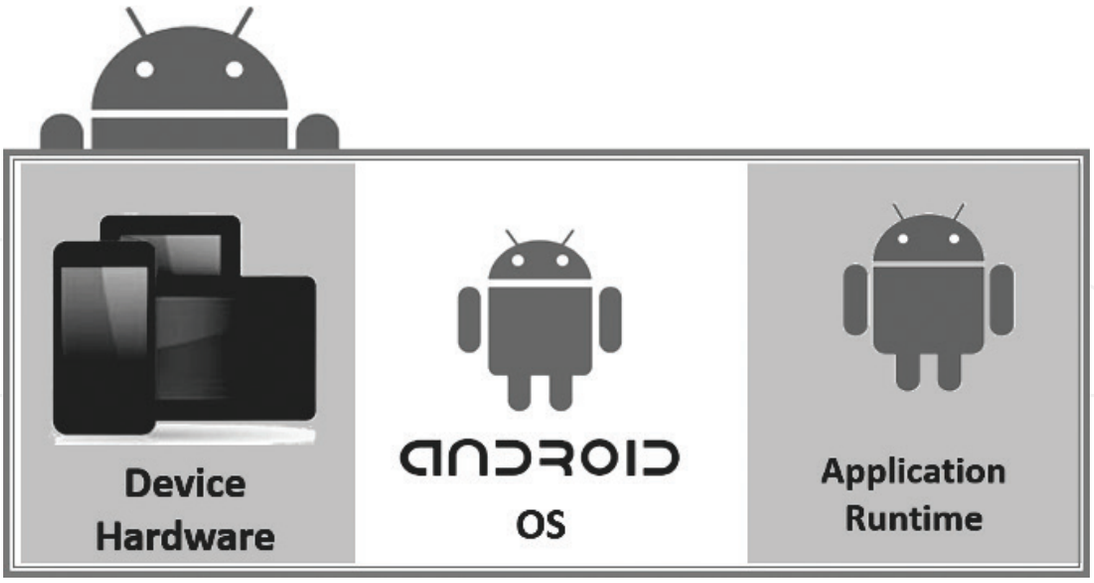

Figure 1. The main Android platform building blocks, adapted from [1].

open platform while still protecting all users of the platform. Security controls were designed to reduce the burden on developers. In this way, security-experienced developers can easily work with and rely on flexible security controls and developers less familiar with security concepts will be protected by safe defaults [1].

Moreover, Android provides a set of key security features, which are: robust security at the OS level through the Linux kernel, mandatory application sandbox for all applications, secure inter-process communication, application signing, and application-defined and user-granted permissions [1]. In the first case, as shown in the Android software stack, see **Figure 2**, each component assumes that the components below are properly secured. In this scheme, with the exception of a small amount of Android OS code running as root, all code above the Linux kernel is restricted by the application sandbox [1]. It is important to notice that the Android kernel is slightly different from a "regular" Linux kernel, the differences are due to a set of features originally added to support Android, and some of them are the low memory killer, wakelocks, anonymous shared memory, alarms, paranoid networking, and Binder. However, Android's security model also takes advantage of the security features offered by the Linux kernel. In a Linux system, which is a multi-user operating system, the kernel can isolate user resources from one another, just as it isolates processes. Consequently, one user cannot access another user's file, unless explicitly granted permission, and each process runs with the identity of the user that started it. In a traditional system, a user ID (UID) is given either to a physical user that can log into the system and execute commands via the shell or to a system service (daemon) that executes in the background. At this point, it is also worth to notice that Android was originally designed without the need for registering different physical users with the system, thus the physical user is implicit and UIDs are used to distinguish applications instead. This forms the basis of Android's application sandboxing [3].

On top of the Linux kernel layer is the hardware abstraction layer (HAL). This layer provides a standard method for creating software hooks between the Android platform stack

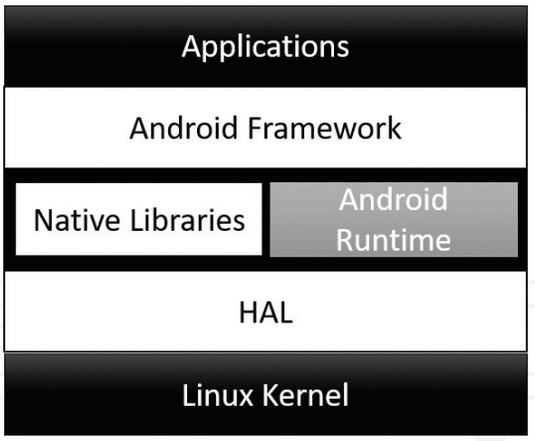

Figure 2. Android software stack, adapted from [1].

and the hardware. The HAL allows to implement functionality without affecting or modifying the higher level system [4]. In the next layer, the libraries component acts as a translation layer between the kernel and the application framework. The native libraries in Android are written in C and C++, most of which are ported from Linux, but are exposed to developers through a Java API. At the same level, there are also components from the Android runtime and core libraries. The virtual machine is an important part of the Android operating system and executes system and third-party applications [4, 5].

At the next level, the Android framework layer provides a suite of services or systems that are useful when writing applications. Commonly referred to as the application programming interface (API) is one of the building blocks for the final system or end-user applications. Finally, at the top most layer, applications component of the Android OS is located, which is the layer closest to the end user. All finished developed products will execute in this space by using the API libraries and the runtime environment [5].

As mentioned before, Android apps are composed of different components and each app is sandboxed by executing it in a separate process with a distinct user ID and assigning it to a private data directory on the file system. The four basic Android application components are *Activities*, *BroadcastReceivers*, *ContentProviders*, and *Services*. All components can be interconnected remotely across process boundaries by using different abstractions of Binder inter process communication (IPC) [6]. These interconnections are commonly referred to as inter-component communication and are the primary communication mechanism in Android although it can provide classical channels such as files or sockets. Android apps can either contact system services or communicate directly with each other. User space processes can communicate with each other over Binder IPC via the Binder kernel module. Android's design provides different levels of abstraction for Binder IPC, allowing developers to easily make use of Binder IPC at the application level to connect different apps' components (stubs, proxies, and managers). All inter-component communication (ICC) is built on the top of Binder IPC [7], see **Table 1**.

Moreover, every application that is run on the Android platform must be signed by the developer. Application signing allows developers to identify the author of the application and to update their application without creating complicated interfaces and permissions.

http://dx.doi.org/10.5772/intechopen.69695

Component	Description
Stubs and proxies	The most basic level of abstraction of Binder IPC. Implement remote procedure calls (RPC) via Binder IPC. A proxy at the caller-side marshals the method parameters into primitive data types and transfers them via IPC to the recipient, where stub unmarshals the primitives into the original parameters and calls the actual method.
System services and managers	Managers are part of the SDK and encapsulate pre-compiled proxies for system apps and services like the location manager service that implement the Android application framework API.
Intents	The highest level of abstraction is so-called intent messages. An intent is a data structure used to provide an abstract description of an operation to be performed by its receiver(s). Common usages of Intents include starting activity components or broadcasting notifications to apps. Since the sender of an intent can both explicitly state the target component and implicitly define potential receivers through a description of the intended action, the actual target app(s) must be resolved at runtime.

Table 1. Inter-component communication (ICC) builds on top of Binder IPC.

Applications that attempt to install without being signed will be rejected by either Google Play or the package installer on the Android device. Application signing ensures that one application cannot access any other application except through well-defined IPC. Applications can be signed by a third-party or self-signed. It is also possible to declare security permissions at the Signature protection level, restricting access only to applications signed with the same key while maintaining distinct UIDs and application sandboxes. A shared application sandbox is allowed via the shared UID feature where two or more applications signed with same developer key can declare a shared UID in their manifest [8].

Android applications can access only their own files and any world-accessible resources on the devices due to the sandboxed nature of Android. However, Android can grant additional, fine-grained access rights to applications in order to allow for richer functionality. Those access rights are called permissions, and they can control access to hardware devices, Internet connectivity, data, or OS services. Applications can request permissions by defining them in the AndroidMAnifest.xml file. At the application install time, Android inspects the list of requested permissions and decides whether to grant them or not. Once granted, permissions cannot be revoked and they are available to the application without any additional confirmation. For some features, explicit user confirmation is required for each accessed object, even if the requesting application has been granted the corresponding permission [3].

Additionally, Android also enforces security by providing preinstalled and user-installed applications. Pre-installed applications work as users applications and as providers of key devices' capabilities that can be accessed by other applications. This application may be a part of the open source Android platform or they may be developed by a device manufacturer for a specific device. On the other hand, Google Play, Android's application official store, offers users hundreds of thousands of applications, including many third-party applications [1].

Outside these security features, Android also provides a set of cloud-based services that are available to compatible Android devices with Google Mobile Services. These services are not part of the Android Open Source Project, but are included on many devices. See **Figure 3**.

Figure 3. The primary Google security services.

Briefly described, *Google Play* is a collection of services that allow users to discover, install, and purchase applications from their Android device or the web. It also provides community review, application license verification, application security scanning, and other security services. The *Android update service* delivers new capabilities and security updates to selected Android devices, including updates through the web or over the air (OTA). The *Application services* term refers to a set of frameworks that allow Android applications to use cloud capabilities such as (backing up) application data and settings and cloud-to-device messaging (C2DM) for push messaging. The *Verify Apps* service warns or automatically blocks the installation of harmful applications, and continually scan applications on the device, warning about or removing harmful apps. *SafetyNet* is a privacy preserving intrusion detection system to assists Google tracking and mitigating known security threats in addition to identify new security threats. The *SafetyNet Attestation* is a third-party API to determine whether a device is CTS compatible. Attestation can also assist to identify the Android application communicating with the application server. Finally, the Android device manager is a Web and Android application to locate lost or stolen devices [1].

As it can be observed from the previous description, Android has become a continuously evolving complex ecosystem composed of multiple subsystems and services that put together an enormous challenge in terms of security. In this context, in the following section, a brief discussion of some attempts to conceptualize and characterize the Android attack surface and key security challenges is presented prior to the later discussion of some of the main malware analysis and detection techniques, as an initial landmark from where techniques and research approaches presented later on may be better referred to or mapped to specific security aspects of the Android ecosystem.

http://dx.doi.org/10.5772/intechopen.69695

3. The android attack surface

An attack surface is a term used to identify the characteristics of a target that makes it vulnerable to attack. An attack vector generally refers to the means by which an attacker performs an attack. In other words, an attack surface refers to the code that an attacker can execute and therefore can attack. In contrast to an attack vector, an attack surface does not depend on the attackers' actions or require a vulnerability to be present, it describes where in code vulnerabilities might be waiting to be discovered. Generally, the size of a target's attack surface is directly proportional to how much it interfaces with other system. Similar to attack vectors, attack surfaces can be discussed both in general and in increasingly specific terms. It is a common result that by studying one particular attack surface, additional attack surfaces are revealed [9].

By focusing on particular risky attack surfaces, a system can be attacked or secured more quickly. Several properties are important when identifying attack surfaces, some of them are: attack vectors, privileged gained, memory safety, and complexity. Because Android devices have such a large and complex set of attack surfaces, it is necessary to divide them [9]. **Figure 4** exemplifies some of the more general attack surfaces for Android devices together with some attack vectors and propagation mechanisms.

The remote attack surface is the largest and most attractive attack surface exposed by an Android device. This name, which is also an attack vector classification, aims to express the fact that the attacker does not need to be physically located near the victim. Instead, attacks are executed over a computer network, usually the Internet. Various properties further divide this surface into distinct groups, see **Figure 4**. The Remote attack surface address the various attack surfaces exposed to code that is already executing on a device. The privileges required to access these attack surfaces vary depending on how the various endpoints are secured. When an attacker has achieved arbitrary code execution on a device, the next logical step is to escalate privileges, either in the kernel space or under the root or system user. The physical attack surfaces give name to the attacks that require physically touching a device, in contrast to physical adjacency where the attacker only needs to be within a certain range of the target. Third-party modification attack surface relates to attack surfaces associated to the modification of various parts of an Android device system, as many parties involved in creating Android devices tend to make extensive changes as a part of their integration process [9].

Unfortunately, on the top of this complexity, Android's security analysis also requires to take into account a set of Android's security challenges such as: fragmentation, malware, management tool selection, user behavior, and compartmentalization [10]. Fragmentation challenge refers to the complexity associated to the wide range of Android-modified versions implemented on different devices. Malware challenge advocates to the rapid increase of malicious applications development and sophistication targeting the Android OS. Management tool selection challenge relates to the selection of management tools, which can avoid overlapping or conflicting features, as well as to maximize IT productivity. The user behavior challenge refers to the need for encouraging users to comply with good security policies and practices. Lastly, compartmentalization describes the challenge of providing dual personal and mobile virtualization, which separates a single device into different personal environments [11].

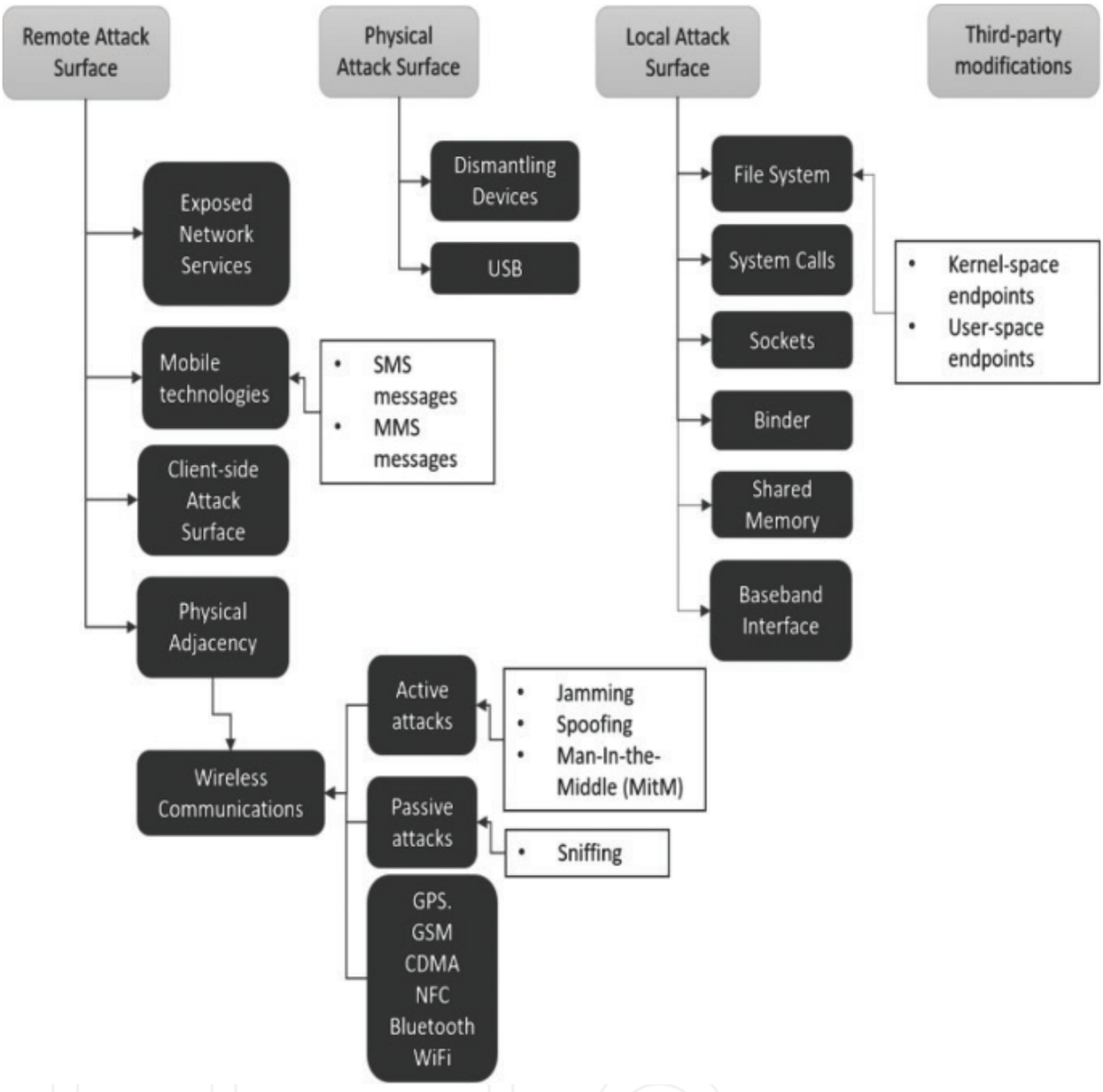

Figure 4. Android's main attack surfaces, based on descriptions in [9].

4. Android malware

Android malware can be characterized in different ways: in [12], a systematic characterization is proposed ranging from their installation, activation, to the carried malicious payloads. Thus, malware installation can be generalized into three main social engineering-based techniques: repackaging, update attack, and drive-by download. Repackaging is one of the most common techniques that malware authors use to piggyback malicious payloads into applications. In essence, malware authors get an application file, disassemble them, enclose malicious payloads, reassemble, and submit the new application to an official or alternative market. Users could be vulnerable by being enticed to download and

install these infected applications. In the case of the update attack, instead of enclosing the payload as a whole only an update component is included, which will fetch or download the malicious payloads at runtime. Because the malicious payload is in the "updated" application, not the original application itself, it is stealthier than the malware installation technique that directly includes the entire malicious payload in the first place. The third technique applies the traditional drive-by download attack to mobile space. Though they are not directly exploiting mobile browser vulnerabilities, they are essentially enticing users to download "interesting" or "feature-rich" applications. This is only a set of common techniques, other threats include combinations of the previous techniques, as well as approaches such as "spyware," which intend to be installed to victim's phones on purpose; fake apps that masquerade as the legitimate applications but stealthily perform malicious actions, such as stealing users' credential; applications that provide the functionality they claimed, they are not fake ones, but that intentionally include malicious functionality, which is unknown to users. At last, a group of applications that rely on the root privilege to function well. The leverage known root exploits to escape from the built-in security sandbox [12].

5. Trends of android malware detection

Malware detection as a discipline combines multiple techniques and principles; Zaki Mas'ud et al. [13] have proposed a general classification including four main categories, see **Figure 5**.

Detection techniques can be classified into three detection techniques: signature-based (SB), anomaly-based (AB), and specification-based (SPB) detection. Signature-based detection refers to the malware detection by comparing the application signature or pattern captured with a database of known attacks or threats. AB detection monitors regular activities in the devices and looks for any behavior that deviates from the normal pattern. Similar to AB detection, SPB detection also monitors for any deviation but rather than detecting the occurrence of specific attack patterns; it monitors for deviation of their behavior from the normal specification. The detection analysis category involves reverser engineering techniques aimed to obtain information about the behavior of a malware in its environment. On the one hand, in static analysis, detection is done through the source code, binary, or the API level without the execution of the Android malware. On the other hand, dynamic detection detects malware through the execution behavior of the malware. In this case, the detection is done through monitoring the execution of Android malware activity at runtime. The detection deployment platform category helps to identify whether the malware detection is deployed in the host or on a remote server. In host detection, all the activity of the device is monitored, analyzed, and processed in the device itself. Meanwhile remote deployment requires a remote server, which monitors the activity of the device on the device but performs the analysis and detection process on the remote server. Another important category is the audit data source monitored in the detection process. The data source collected in the Android malware detection can be traced within the five Android framework layers (i.e. application, application framework, Android runtime, libraries, and Linux kernel layers). In addition, network traffic data can also be monitored for any malicious communication activity through the network [13]. Multiple

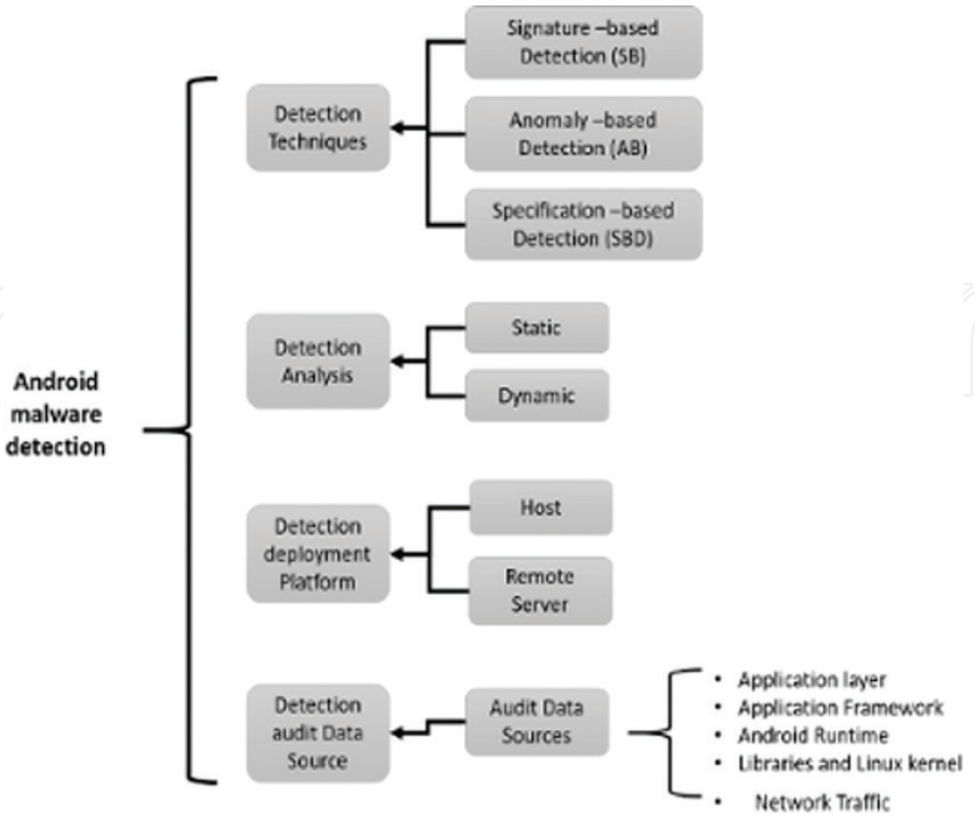

Figure 5. Classification of Android malware detection approaches.

researchers have analyzed different approaches; **Figures 6** and **7** provide an overview, based on the descriptions presented in [14], of different features and algorithms utilized for static and dynamic malware analysis in different research works.

Nowadays, most detection techniques for Android malware use statically extracted data from the AndroidManifest.xml file or Android API function calls, as well as dynamically obtained information from network traffic and system call tracing [15]. Moreover, most current detection systems equipped with a database of regular expressions that specify byte or instruction sequences that are considered malicious are largely based on syntactic signatures and employ static analysis techniques. Unfortunately, static and signature-based analysis techniques can be evaded by malware applications using techniques, such as polymorphism, metamorphism, and dynamic code loading [16].

Dynamic analysis defines a set of rules for the application behavior, which are challenged for an application according to a possible attack. An event is simulated for each rule and the triggered behaviors are checked to detect malware applications. In some cases, modern malicious applications are capable to evade dynamic analysis as they become aware of the analysis environment, or due to the inability of the malware sample to obtain some required external data or service [16].

As security threats evolve, static and dynamic analysis techniques are less capable to identify malware code by their own. Thus, hybrid approaches combine aspects of both static and dynamic

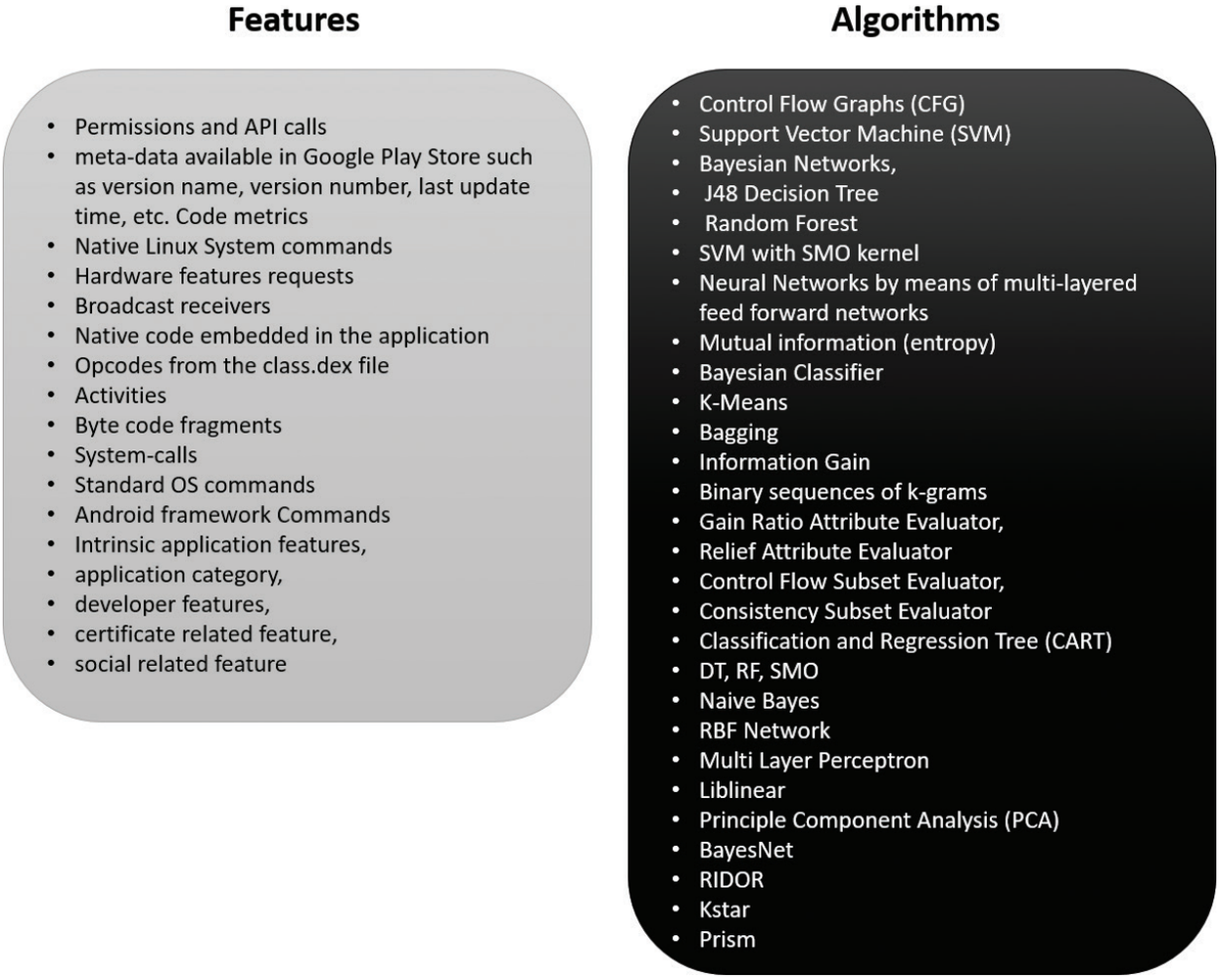

Figure 6. Some common static analysis features and algorithms that are used to process them for different research approaches, based on [14].

analysis [17]. The implementation of hybrid solutions for malware analysis and detection is not a new approach in the PC anti-malware literature [18]. However, the particular characteristics and constraints of mobile devices have defined a new area for their own. In this sense, for example, even when malware analysis and detection schemes can be deployed either on a local basis or offloaded to an external equipment, like a remote server, differences between the mobile and PC ecosystems imply a totally different approach to solve this challenge in both cases. In the particular case of mobiles devices, most current client side security solutions include anti-virus or anti-malware applications installed on the devices to protect them against known applications installed on the mobile devices based on known signatures of malicious applications [19]. On the other hand, cloud-based systems are mainly designed to offload a significant part of the operation to the cloud. Both approaches entail performance constraints and disadvantages. As an example, in applications installed on mobile devices aiming to provide real-time protection, there is an associated decrement in the device's performance and battery life, while cloud-based approaches making use of high end resources cannot offer real-time protection by their own, as they can leave devices vulnerable when connectivity with the server is poor [20].

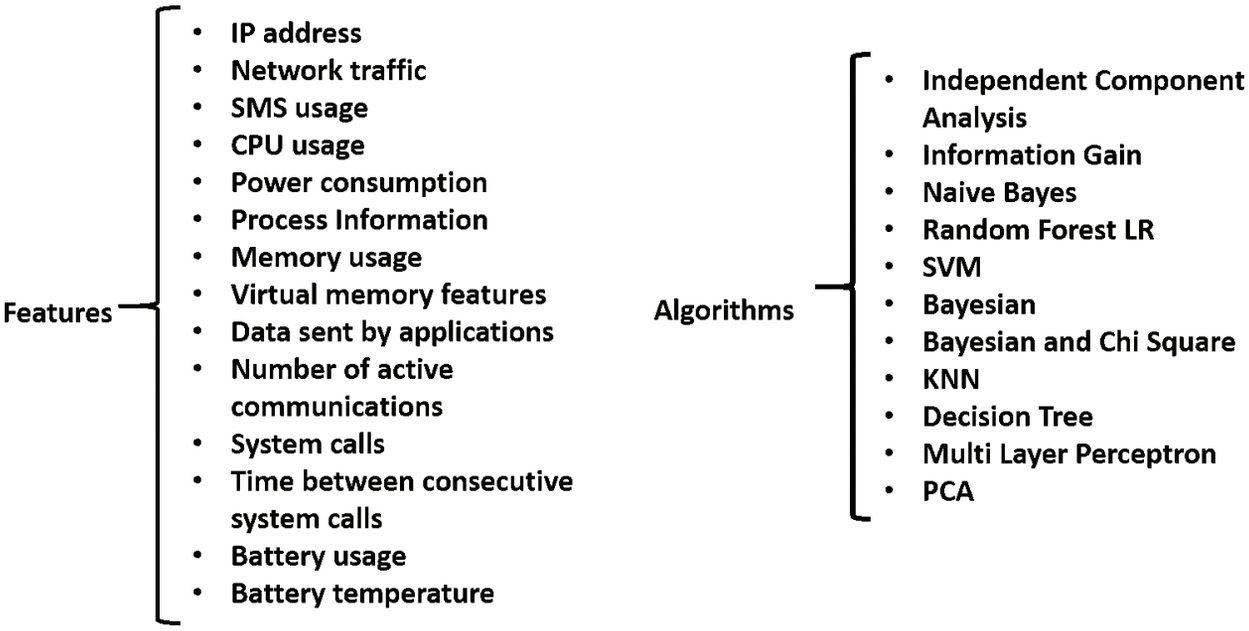

Figure 7. Some common dynamic analysis features and algorithms that are used to process them for different research approaches, based on [14].

Unlike hybrid detection and analysis schemes taking advantage of both static and dynamic analysis, as well as from local and remote combined implementation or execution, are generally common for PC equipment, these schemes are not common for mobile devices. Most solutions combine static and dynamic analysis methods or local and remote deployments but not both of them, as this would require too many compromises to be achieved with the current technologies [18].

6. Conclusions

The continuous development and fast change of the smart devices market has promoted an increase in the number of services and applications offered. As these devices integrate to the users every day activities, they become very attractive targets for cyber criminals. In this sense, malicious software (malware) has become a main security issue in this area. Although malware is not a new problem in the IT industry, differences between PC and smart devices make smart devices security a different problem bounded to the particular features of mobile devices. Moreover, the big number of stakeholders ranging from device manufactures to communication service providers creates a highly heterogeneous environment where attack surfaces characterization becomes a very complex task. In this context, this chapter aimed to present an overview of the fundamental aspects for Android malware analysis and detection.

As it can be deduced from the information discussed above, generally speaking there is a core set of analysis techniques and resource data that have been utilized in multiple research approaches in order to identify and detect malicious software. This observation may be obvious as the identified features are core elements of the Android security architecture,

although there is not full agreement in the best technique or procedures for effective malware detection. It is important to notice that machine learning has an important role in most of the discussed approaches and in the state of the art for malware analysis, and in some cases, reported results look highly promising, but there is always the problem of having a limited number of samples to test all possible threats. Additionally, with the current vast set of analysis and reverse engineering set of tools, which are implemented under different technologies and analysis approaches, integration seems a very difficult task to achieve. Moreover, different tools provide multiple and different levels of automation. However, a need for automating most of the process is still an important issue as most of the analysis in the identification of new threats continues to be a human task.

Finally, it is expected that the information presented in this chapter would help readers to obtain a general view of the Android malware analysis and detection area from where she or he can visualize new avenues of research.

Acknowledgements

The authors acknowledge the support from the Mexican Consejo Nacional de Ciencia y Tecnología (CONACYT) under grant number 216747 and Instituto Politécnico Nacional-Secretaria de Investigación y Posgrado under grant numbers 1894 and 20170344.

Author details

Abraham Rodríguez-Mota[1,3]*, Ponciano J. Escamilla-Ambrosio[2] and Moisés Salinas-Rosales[2]

*Address all correspondence to: arodrigm@cic.ipn.mx

1 ESIME Zacatenco, Instituto Politécnico Nacional, Mexico City, Mexico

2 CIC, Instituto Politécnico Nacional, Mexico City, Mexico

3 Centro de Investigación en Computación, Laboratorio de Ciberseguridad, Mexico City, Mexico

References

[1] Android. Security [Internet]. Available from: https://source.android.com/security [Accessed: 12-12-2016]

[2] Android. ART and Dalvik [Internet]. Available from: https://source.android.com/devices/tech/dalvik/ [Accessed: 02-10-2017]

[3] Elenkov N. Android Security Internals. An in-Depth Guide to Android's Security Architecture. USA: No Starch Press; 2015 401 p

[4] Android. Android Interfaces and Architecture [Internet]. Available from: https://source.android.com/devices/ [Accessed: 20-02-2017]

[5] Gunasekera SA. Android Apps Security. New York: Apress; 2012 2021 p

[6] Backes M, Bugiel S, Gerling S. Scippa: System-centric IPC provenance on Android. In: 30th Annual Computer Security Applications Conference; December 08-12-2014; New Orleans, Louisiana, ACM, New York, NY, USA; 2014. pp. 36-45. DOI: http://dx.doi.org/10.1145/2664243.2664264

[7] Enck W, Ongtang M, McDaniel P. Understanding android security. IEEE Security & Privacy. 2009;**7**(1):50-57. DOI: 10.1109/MSP.2009.26

[8] Android. Application Signing [Internet]. Available from: https://source.android.com/security/apksigning/index.html [Accessed: 03-05-2017]

[9] Drake JJ, Fora PO, Lainer Z, Mulliner C, Ridley SA, Wicherski G. Android Hacker's Handbook. USA: Wiley; 2014

[10] Mathias C./TechTarget. Top five Android device management security challenges [Internet]. Available from: http://searchmobilecomputing.techtarget.com/tip/Top-five-Android-device-management-security-challenges [Accessed: 02-02-2016]

[11] Rodríguez Mota A, Escamilla Ambrosio PJ, Aguirre Anaya E, Acosta Bermejo R, Villa-Vargas LA. Improving Android mobile application development by dissecting malware analysis data. In: 4th International Conference in Software Engineering Research and Innovation; 27-29 April 2016; Puebla, Mexico. IEEE; 2016. pp. 1-6. DOI: 10.1109/CONISOFT.2016.21

[12] Zhou Y, Jiang X. Dissecting Android malware: Characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy. 2012. DOI: 10.1109/SP.2012.16

[13] Zaki Mas'ud M, Dahib S, Abdollah MF, Selamat SR, Yusof R. Android malware detection system classification. Research Journal of Information Technology. 2014;**6**:325-341. DOI: 10.3923/rjit.2014.325.341

[14] Baskaran B, Ralescu A. A study of android malware detection techniques and machine learning. In: Phung PH, Shen J, Glass M, editors. Modern Artificial Intelligence and Cognitive Science; 22-23 April 2016. Dayton, OH, USA: CEUR; 2016. p. 15-23

[15] Afonso VM, Favero de Amorim M, Grégio ARA, Barroso Junquera G, Lício de Geus P. Identifying android malware using dynamically obtained features. Journal of Computer Virology and Hacking Techniques. 2015;**11**(1):9-17. DOI: 10.1007/s11416-014-0226-7

[16] Moser A, Kruegel C, Kirda E. Limits of static analysis for malware detection. In: Twenty Third Annual Conference in Computer Security Applications; 12 2007; Miami Beach, FL, USA. IEEE; 2007. DOI: 10.1109/ACSAC.2007.21

[17] Damodaran A, Di Troia F, Vissagio CA, Austin TH, Stamp M. A comparison of static, dynamic, and hybrid analysis for malware detection. Journal of Computer Virology and Hacking Techniques. 2017;**13**(1):1-12. DOI: 10.1007/s11416-015-0261-z

[18] Rodríguez Mota A, Escamilla Ambrosio PJ, Morales Ortega S, Salinas Rosales M, Aguirre Anaya E. Towards a 2-hybrid Android malware detection test framework. In: International Conference on Electronics, Communications and Computers (CONIELECOMP); Cholula, Puebla, México. 2016. pp. 54-61. DOI: 10.1109/CONIELECOMP.2016.7438552

[19] Penning N, Hoffman M, Nikolai J, Wang Y. Mobile malware security challenges and cloud-based detection. In: International Conference on Collaboration Technologies and Systems (CTS); Minneapolis, Minnesota. IEEE; 2014. pp. 181-188. DOI: 10.1109/CTS.2014.6867562

[20] Damopoulos D, Kambourakis G, Portokalidis G. The best of both worlds: A framework for the synergistic operation of host and cloud anomaly-based IDS for smartphones. Seventh European Workshop on System Security (EuroSec'14). 2014;**6**:1-6:6. DOI: 10.1145/2592791.2592797

IntechOpen