AMon: A Monitoring Multidimensional Feature Application to Secure Android Environments

J.A. Gómez-Hernández
Dept. Languages and Computer Systems
Granada, Spain
jagomez@ugr.es

P. García-Teodoro
Dept. Signal Theory, Telematics and Communications
Granada, Spain
pgteodor@ugr.es

J.A. Holgado-Terriza
Dept. Languages and Computer Systems
Granada, Spain
jholgado@ugr.es

G. Maciá-Fernández
Dept. Signal Theory, Telematics and Communications
Granada, Spain
gmacia@ugr.es

J. Camacho-Páez
Dept. Signal Theory, Telematics and Communications
Granada, Spain
josecamacho@ugr.es

M. Robles-Carrillo
Dept. International Law and International Relations
Granada, Spain
mrobles@ugr.es

***Abstract*—. This work introduces a novel Android monitoring app named *AMon*. It is aimed at collecting device related information from several sources: communications, */proc* filesystem, applications and device usage. The information is dynamically gathered over time and its execution does not require to get special privileges or to be system root. In order to assess *AMon* capabilities, we have used it as the acquisition module of a subsequent security incident detection process. The results obtained show a good performance in terms of battery consumption, CPU and RAM usage, as well as overall system overhead. In order to contribute to the community, *AMon* is available at a public repository for free use and improvement.**

Keywords—monitoring, mobile security, anomaly detection

I. INTRODUCTION

Mobile devices such as smartphones and tablets are the most accepted platforms among users nowadays worldwide. According to GSMA [1], there are around 3.6 billion mobile users at present (60% from smartphones) and it is expected this number will increase to around 5 billion in 2025. This constitutes around 29 Exabytes of traffic per month nowadays and around 80 Exabytes per month in the next years [2].

Provided the increasing relevance of mobile devices, mobile malware has also experienced a huge increment [3]. Moreover, according to the fact that around 87% of the mobile market corresponds to Android devices [4], this OS is exposed to a number and variety of threats and attacks [5].

Automatic anti-malware solutions for mobile devices are based on some kind of detection procedure to notify about the observation of undesired activities or behaviours [6]. For that, a monitoring process to gather and analyse specific operational information is required.

This work contributes a novel Android monitoring tool able to be used in mobile security dynamic detection proposals. It is named *AMon* (standing for 'Android Monitoring') and is aimed to collect information about a number of aspects such as communications, apps, security state, and interfaces state. Moreover, we must remark that *AMon* does not require special permissions or root access to operate and that it collects a broader set of characteristics than others found in the literature with a low computational cost.

The organization of the rest of the paper is as follows. Section II presents main proposals in the field of mobile monitoring and detection in the specialized literature. Section III describes *AMon* from a technical perspective, its capabilities being detailed. In Section IV, *AMon* is used as the gathering module of a detection ML system with security purposes, the results obtained being discussed. Finally, main conclusions and remarks are presented in Section V.

II. STATE OF THE ART

The number of detection solutions in the literature to fight against security events has grown in the last years [7][8], where a variety of analysis techniques are considered.

A principal issue regarding malware detection is monitoring, as it allows collecting the parameters, variables and/or activities that will represent the state of the target system over which to subsequently decide about its benign or malicious nature. For instance, TaintDroid [9] tracks the flow of privacy sensitive data through third-party applications, as it assumes that downloaded, third-party applications are not trusted. CrowDroid [10] also performs a dynamic detection, collecting data regarding basic device information, installed applications list and the result of monitoring applications with *strace* tool to collect system calls.

Instead, DREBIN [11] was proposed to perform a broad static analysis, gathering as many features from an application's code and manifest as possible. These features are organized in sets of strings (such as permissions, API calls and network addresses) and embedded in a joint vector space. Authors in [12] also introduce a static detection proposal. The extraction of the features is automatically performed here by using scripts in order to get the permissions, intents, hardware and software features such as API calls and network access.

After the previous proposals, additional acquisition systems have been developed. In [13], the authors propose a network traffic monitoring system to detect Android malware. The system parses the protocol of data packets and extracts a set of 10 features (ID process, start and end of connection time,

DOI 10.1109/SPW53761.2021.00013

upward and downward flows, source and destination IPs, source and destination ports, protocol type), then uses an SVM classification algorithm for data classification, determine whether the network traffic is abnormal, and locate the application that produced it through the correlation analysis.

Authors in [14] make use of 59 features in total related with memory (24: Active, Buffers, MemFree, AnonPages, etc.), CPU (10: User, Nice, System, CPU usage, etc.) and network (25: InReceives, InMsgs, InSegs, EstaResets, ActiveOpens, etc.) mainly extracted from the */proc* folder for Android malware detection.

In [15], a method that uses System Flow Graphs is proposed to detect the execution of Android malware by monitoring the information flows they cause in the system. For that, authors make use of AndroBlare to track information flows between files, processes and sockets occurring on Android. Also related with graphs, authors in [16] develop an automatic Android malware detection system, Deep4MalDroid, using deep learning framework based on the Linux kernel system call graphs.

In [17], BehaviourDroid is presented. It is a robust, extensible and configurable framework that can simultaneously monitor multiple apps and properties from loggings and system calls.

As in the case of DREBIN, authors in [18] propose the use of intents raised by applications as a metric to identify the malicious behaviour of an application. An intent is a request from an application for performing a certain action, the main purpose of intent in Android platforms another activity, that activity generates an intent and encapsulates the required information that it wishes to communicate.

In [19], Feng *et al.* introduce EnDroid, an effective dynamic analysis framework aimed to implement highly precise malware detection based on multiple types of dynamic behaviour features. These features cover system-level behaviour trace and common application-level malicious behaviours like personal information stealing, premium service subscription, and malicious service communication.

The use of permissions and API function calls is also considered to detect malware in Android in [20]. Previously, authors complemented these features with runtime features such as user operations [21]. Similarly, Koli in [22] makes additional use of app's key information like dynamic code, reflection code, native code, cryptographic code and database from applications.

In [23], authors design and implement BDfinder, an easy-to-deploy and dynamical passive backdoor detection system to detect and visualize backdoor behaviours in real time. For that, system calls and binder data are considered to build sensitive behaviours.

In this overall context, we introduce here *AMon*, a novel open-source monitoring tool intended to collect information from Android devices with several purposes, in particular to secure the environment. *AMon* allows to gather multidimensional dynamic information related with network communication, applications, hardware resources, and protection mechanisms of devices where it has been installed. Our tool collects a broader set of characteristics than others found in the literature with a low computational cost. Moreover, community can publicly contribute to *AMon* by adding new monitoring capabilities.

III. MULTIDIMENSIONAL MOBILE DEVICE MONITORING

AMon is a JAVA tool oriented to data gathering from multiple sources in Android platforms. For this, most of the functionality is implemented by using the Android API for developers [24]. However, Android is an OS that is continuously updated, implementing new functionalities and providing changes from one version to another. Moreover, *AMon* operation relies on the accessibility to system information, which is difficult and different to obtain depending of the Android version, especially if we consider that *AMon* does not require root access. Therefore, the selection of the the minimal SDK version is not a trivial decision.

In Android *Oreo* the access to network data via folder */proc/net* was disabled. Indeed, the access to */proc* pseudo-filesystem was heavily limited. This leads the inability to get CPU stats for future versions, but network data can be accessed via a workaround that uses a local VPN.

Thus, *AMon* is developed with Android *Oreo* as target and is compatible with Android *Pie*. The current minimal supported version is Android *Marshmallow*, but it could be lowered at the cost of some functionalities like getting the device IP address.

Due to the broad scope of *AMon* regarding data gathering, its functionality is distributed around four modules as follows.

The *Communications module* implements a local VPN to track network traffic and get data and statistics from it. This module is based on the NetGuard project [25], a firewall application able to prevent applications from connecting to the Internet, logging traffic, outputting to a PCAP file or getting traffic usage. For the *AMon* purpose, the traffic logging capacity has been restructured to get only header information in addition to some statistical information that will be discussed below. Furthermore, this module also includes utilities to get the current IP address (Android 6.0+) and the MAC address.

The *Application module* provides a list of the installed applications on the target device, their versions, and the permissions declared in the manifest file. Also, the module takes a timestamp every time a given app is updated or installed.

The *Hardware resources* module brings access to the state of the device. CPU, RAM and battery values use, both current and maximum, are accessible through this module, as well as the state of the communications such as Wi-Fi, Bluetooth, Mobile Data, GPS interfaces.

The *Protection mechanisms* module makes possible the access to some protection checks such as if the device is rooted, if a pattern is used to lock the device, and if *UnknownSources* or *DeveloperOptions* are enabled.

A more detailed description about each functional module and its implementation is discussed in the following subsections.

A. Traffic measurement

The *Communications module* functionality is distributed between a library inherited from NetGuard and other that

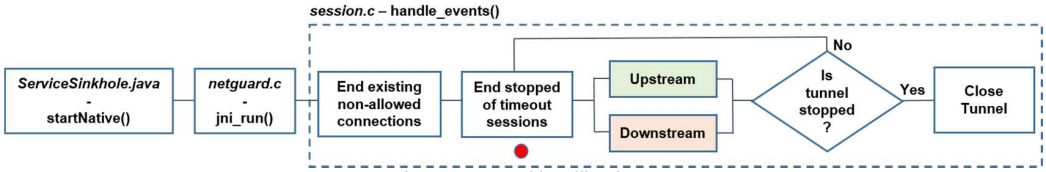

Figure 1. Tunnel handling in C.

provides additional capabilities related to networking. Due to the main functionality inherited from NetGuard, it will be discussed firstly.

As previously indicated, the decision to use a local VPN comes from the need to get networking data in newer Android versions. After Android 8.0+, the access to network information is only possible through a VPN. So, the app is responsible for handling the traffic quickly enough to don't hinder the user work and don't drain resources of the system.

Several implementations create the local VPN purely in Java, which handle data in an easy way due to be all structured in classes and with access to high level functionalities. However, the performance is degraded. The alternative is to use native code in C/C++ with the NDK toolset. For that, JNI is used to provide the interface of interoperability between Java and C/C++. The NetGuard application follows this path and includes a specific C/C++ library. For its use in *AMon*, the associated internal operation will be explained below. With this purpose, the process will be breakdown in three steps.

The first step (Figure 1) starts at the `ServiceSinkhole` class in Java, where the VPN is created and passed as a reference to the C/C++ code. Once a packet arrives to the native code processing, it goes from *netguard.c* to the `handle_events()` function at *session.c*, which contains the main procedure. After ending previous existing connections, which are not already allowed, a loop starts. This loop checks for connections that need to be terminated (due to a time-out or an explicit stop) and contains the logic for the upstream and the downstream paths. This loop ends when the tunnel is forced to stop.

The upstream path, represented in Figure 2(a), shows how outgoing connections are handled. This path follows to `check_tun()` at *ip.c*, where we check if the packet is allowed to be forwarded, it being discarded if not allowed. Then, the protocol is labeled as TCP, UDP or ICMP. Each one leads to its own file that handles the rest of the communication, finally forwarding the packet from the mobile device to the Internet.

The last step, Figure 2(b), represents the downstream path handling incoming connections to the device. The process followed in this path is more complex that the corresponding to the upstream one. For TCP, it starts checking the socket state: the SOCK5 process is performed in case of listening, the data being forwarded otherwise. After that, the socket is checked for a response from the other side: if some data arrives, it is passed to a buffer to be consumed by the device. For UDP and ICMP protocols, the same scheme applies: incoming data is checked at the socket and the received data is written to the buffer.

The implementation of the NetGuard app includes some points where data is gathered and collected to then report this information to the Java counterpart for data usage statistics or logging. Figure 2 shows this points in colored circles: *green*→ start of flow tracking; *red*→end of flow tracking; *blue*→ outgoing data log; *magenta*→incoming data log; and, *orange* →write *pcap* file if enabled.

Due to the *AMon* ability to get additional netflow related information (source and destination IP addresses, source and destination ports, IP protocol, sent and received bytes, sent packets and packets, TCP flags, ToS, start time and duration), the code contains some probes to acknowledge such data.

AMon also adds a periodic control of the flow respect to the NetGuard implementation. The original functionality only considered the data sent to the Java part at the end of the flow life. Now, after ending the flow check at the start of the connection loop, a verification point is added to acknowledge when a specific time has elapsed since it started.

The information gathered at C level is packed in Java objects for further treatment at Java code. NetGuard uses several Java objects in C to share data with Java or check allowance of a flow. The base Java objects are `Packet`, `Allowed`, `RR` (Resource Record) and `Usage`. *AMon* adds the `Flow` object with the gathered data and the `uid` of the process that originated the connection.

When a flow ends or the periodic check is triggered, the Java method `captureFlow()` is called from C, where a *Flow* object is passed. Those requests to the method are handled by a *Handler* (`LogHandler` class defined in `ServiceSinkhole`). This implementation mimics the one used by NetGuard, allowing further compatibility. When the handler receives a message, the `flow()` method is called, which adds to the Flow object the name of the application using the stored `uid`. Also anonymization occurs here, the application name and the source address being modified if activated. The resulting flows are stored in a queue. The original implementation uses the SQLite3 database, but in our case, due to the high rate of data generated, the continuous access to the database will hinder the user experience. Thus, an in-memory queue is used, which is periodically dumped on a server. For other implementations, a change is necessary for adaptation.

In addition to flow capture, *AMon* is also packed with some utilities to provide access to the MAC address of the device and the current IP address. MAC address is obtained from the network interface information and it is parsed to a string (method available for Android 6.0+). The method responsible to obtain the IP address is limited to Android 6.0 + and gets the information through the *ConnectivityManager* function.

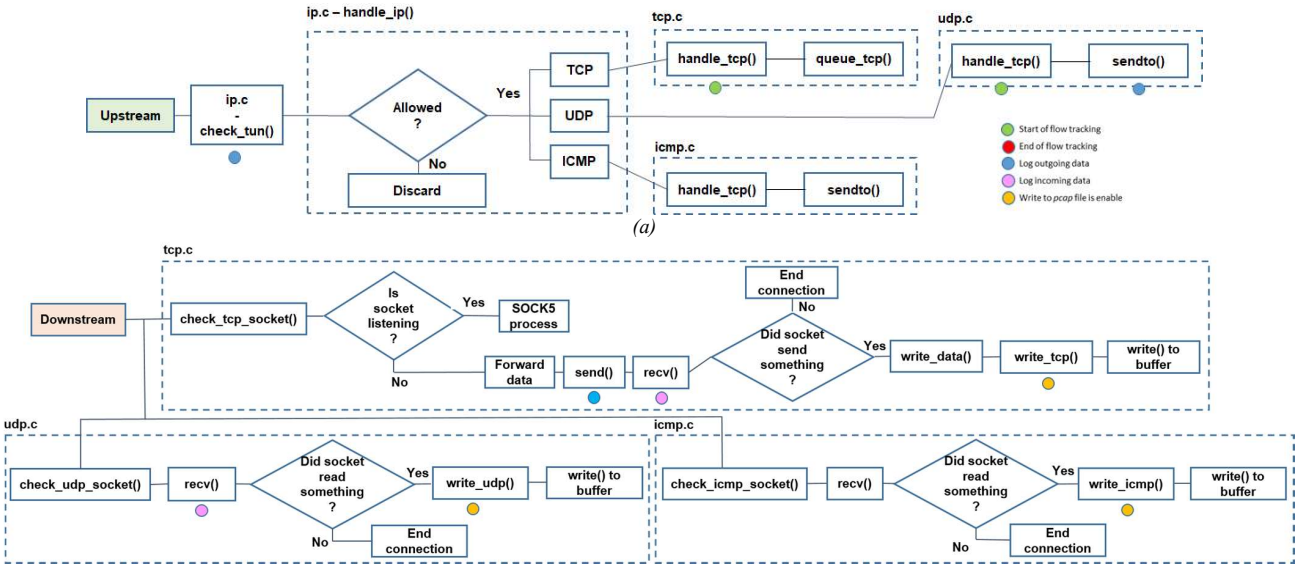

Figure 2. Packet handling in Amon: outgoing packets (a) and incoming packets (b).

B. *Installed Applications*

The *Applications module* is focused on getting the installed apps on the device and related information about them. Apart from the name of the app, *AMon* also gathers the package name, its version and the permissions associated to each of them. It is important to note that our app stores all the possible permissions that the app can request, and not necessarily the ones granted by the user. This information can help to detect undesired behaviours due to the permissions requested or forged applications.

A raw storage of the data would take too much space, mainly due to the permissions list. In order to solve this, a bit encoding has been used to minimize the space requirements. Each permission is listed as 1 if present, and 0 otherwise. The list of permissions used as reference is located at Android Developer for the API level 29. This list is heavily dependent on the API used as target due to the inclusion of additional permissions in newer versions.

C. *Hardware resources*

This module is intended to retrieve hardware related information, which can be split in three groups: static data and specifications; usage of the device; and state of the communication interfaces.

The first category encapsulates the technical information associated to the device. A part of the data is obtained from the `Build` class shipped in the Android API, which provides the SDK used, the brand, the model and the manufacturer, among other data. More technical information is about the number of CPU cores, the RAM size and the battery capacity. Obtaining this last variable is not a simple task, since it requires the use of reflection. The battery capacity in mAh is returned from the `getBatteryCapicity()` method of the class `com.android.internal.os.PowerProfile`. Since this method is not in the public API, the accessibility to it in a future API is not assured and could be changed without notice.

The information about the usage of the device includes the RAM percentage usage, the CPU percentage averaged among all the cores, and the battery level. RAM and battery related information is supported by the public API, but the CPU consumption is not easy to obtain. The usual method implies the access to the */proc/stat* file, but for Android *Oreo* this file and functionality is not available.

In order to retrieve the state of the communication interfaces, Android provides public methods in the API. These sections encapsulate information about the state of the WiFi, the Bluetooth, the location service, the mobile data and the airplane mode. Usual checks of availability are handled by a call to the API, but the mobile data check is not so easy for previous versions to Android 8. For older versions the use of reflection is needed, so the call would be done through the method `getMobileDataEnabled` in the `ConnectivityManager` class. Location services include the location by GPS and the one used by the network. Both of them are supported by *AMon*. In addition, a list of configured WiFi networks is stored in the device, along with the associated security protocol, and a list of bounded Bluetooth devices are also accessible. Furthermore, checking if the current WiFi connection is secure is also included. Note that for the last checks, the associated service needs to be turned on.

D. *Protection mechanisms*

This last module provides access to simple checks about some basic security mechanisms. The absence of these mechanisms is not a security fault but can imply a risk. The checked options are: Unknown Sources; Developer Options;

PIN, pattern or password in the lock screen; and Availability of the root user.

Unknown Sources is a system option in previous versions that allows (or not) the installation of an *apk* file from external sources. After Android *Oreo*, this has changed and is an app-wide option. Thus, for new versions a permission check is used instead. The permission REQUEST_INSTALL_PACKAGES would allow an application to install external *apk*.

Checking the *Developer Options* or the existence of a method to *lock screen* is done by using the public API. Note that the specific locking method used can't be discerned. The last security check is provided by the *RootBeer* library.

IV. EXPERIMENTAL RESULTS

After describing *AMon*, in this section we make use of it to feed a detection tool to determine potential harmful events in mobile environments.

Although several detection approaches can be considered with this purpose we use here MSNM (Multivariate Statistical Network Monitoring), which is an interpretable machine-learning detection methodology introduced by authors in [26]. MSNM presents two relevant benefits: *diagnosis capabilities*, to determine the real causes of inadequate behaviours, and *privacy compliance,* as no private information needs to be accessed by or distributed to third parties.

A. Experimental data

From the above, the specific experimentation environment deployed is as follows. First, a total of 83 final mobile devices have been monitored during 205 days to collect a number of features regarding their associated configuration and communication profiles as specified in Section III.

It is worth to mention that the mobile devices involved in experimentation belong to volunteer users who have signed an agreement to allow the monitoring process to obtain the mentioned individual data for scientific purposes.

Based on the data gathered, MSNM performs two successive stages to analyze the behaviour of the monitored devices: *(i)* first, the behavioural model *P* is estimated from a training dataset (that it is a subset of monitored devices); *(ii)* second, every device is monitored to obtain the associated security profile $<D\text{-}st, Q\text{-}st>^t$, *D-st* and *Q-st* being behaviour related statistics, and, from that, to determine potential deviations in its behaviour with respect to *P.* If so, users will be notified and access restrictions could be applied.

B. Results

In the first stage, for training purposes, we consider the data corresponding to the first 139 days monitored by *AMon*, which correspond to a total of 67 trustable devices.

During this process, some mobile devices exhibit anomalous behaviours. In particular, six of them, those labeled as *D428*, *D1*, *D25*, *D860*, *D394*, and *D190*, present higher *Q-st* and/or *D-st* values than the normality thresholds established and shown in Figure 3 through dashed lines.

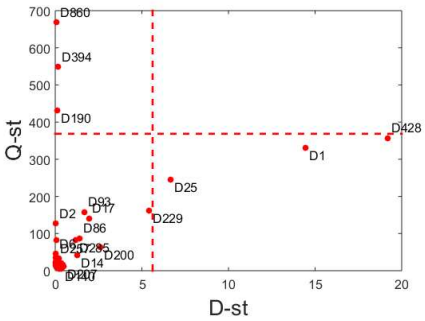

Figure 3. *D-st* and *Q-st* values for devices used in training, where dashed lines represent the normality threshold values.

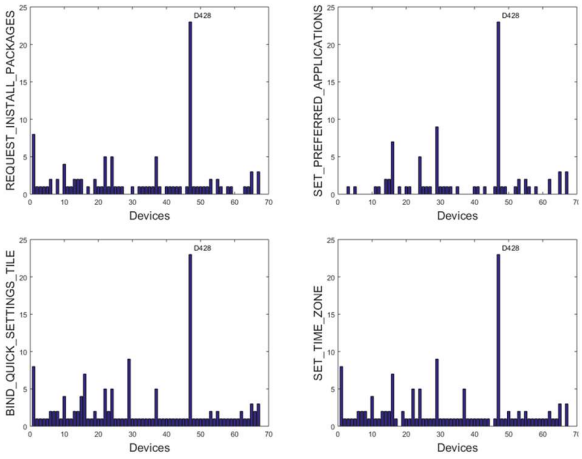

Figure 4. Number of apps using four of the most significant permissions of the anomalous device *D428* in training stage.

After diagnosing the anomalies with oMEDA, we observe deviations in the permissions list per device. A more detailed diagnosis of the device with the most anomalous behaviour, *D428*, reveals significant deviations in 36 of a total of 158 app permissions monitored in comparison with normal devices. Figure 4 shows the number of the four most used permissions: REQUEST_INSTALL_PACKAGE, SET_PREFERED_ APPLICATION, BIND_QUICK_SETTING, and SET_TIME_ZONE. In particular, there is a huge difference in the usage of REQUEST_INSTALL_PACKAGE in *D428* with respect to the rest of devices. This permission is considered dangerous because it is related with malware spreading [27].

With the previous diagnosis, we conclude that the *D428* device does not meet the security requirements needed. With same diagnosis procedure, we determine that the other 5 outliers found (Figure 3) are also below security requirements.

After removing the outliers, all the devices in the network meet the expected security levels. From them, the 'normality' model *P* is estimated, which will be subsequently used to evaluate observed devices against *P* over time. For this experimentation, the gathered activity of all the 83 available mobile devices during the whole sampling period is analysed.

At this point, after analysing *D* and *Q* statistics, device *D116* exhibits a notoriously anomalous behaviour (very far from the

control limits) regarding configuration profiles. We conclude that the cause of such an anomaly is the WRITE_SYNC_SETTINGS permission. The threat associated with it is due to the possibility of data synchronization with external sources.

Regarding communication profiles, the analysis of *D-st* and *Q-st* for daily traffic samples shows the existence of days with very clear anomalous behaviour. After diagnosing the anomaly with oMEDA, we conclude that one of them is motivated by very low traffic, which is not usually a real problem. However, another anomaly is due to *BitTorrent* traffic generated by the device *D473*, which can constitute a security risk depending on the security policy of the access provider. Likewise, the anomaly appeared around 130-*th* day, is due to traffic from devices *D260* and *D1*, which involve an abnormal amount of *NetBios* related traffic.

From the above, the causes detected as anomalies could be used to restrict the access to the devices to our infrastructure at a given instant. However, the final users could be firstly notified to solve the problem and, thus, strengthen global security.

V. CONCLUSIONS

In this paper, we introduce *Amon*, a novel multidimensional monitoring tool for Android devices. Through *Amon* we are able to gather information regarding communications, applications and permissions, usage of device resources, and configuration of the device.

AMon is used to feed a machine-learning detection tool. The results obtained show that the information collected by it are useful for detection purposes. Moreover, *AMon* source code at https://github.com/nesg-ugr/AMon, which is of high interest for the research community.

ACKNOWLEDGMENT

This work has been partially supported by Spanish Government-MINECO and the ERDF (European Regional Development Fund) through project TIN2017-83494-R.

REFERENCES

[1] GSMA: "The Mobile Economy 2019". GSMA, 2019. Available at https://www.gsmaintelligence.com/research/?file=b9a6e6202ee1d5f787cfebb95d3639c5&download

[2] Cisco: "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2017–2022 White Paper". Cisco, 2019. Available at https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-738429.html#_Toc953327

[3] Kaspersky: "IT threat evolution Q2 2018. Statistics". Report, 2018. Available at https://securelist.com/it-threat-evolution-q2-2018-statistics/87170/

[4] IDC: "Smartphone Market Share". IDC report 2019. Available at https://www.idc.com/promo/smartphone-market-share

[5] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M.S. Gaur, M. Conti, M. Rajarajan: "Android Security: A Survey of Issues, Malware Penetration, and Defenses". IEEE Communications Surveys & Tutorials, vol. 17, n. 2, pp. 998-1022, 2015.

[6] Kaspersky: "Android Mobile Security Threats", Report, accessed on July 2019. Available at https://www.kaspersky.com/resource-center/threats/mobile

[7] P. Yan, Z. Yan: "A survey on dynamic mobile malware detection". Software Quality Journal, vol. 28, pp. 891-919, 2018.

[8] Y.S.I. Hamed, S.N.A. AbdulKader, M.S.M. Mostafa: "Mobile Malware Detection: A Survey". International Journal of Computer Science and Information Security, vol. 17, n. 1, pp. 1-10, 2019.

[9] W. Enck, P. Gilbert, B.G. Chun, L.P. Cox, J. Jung, P. McDaniel, A.N. Sheth: "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones". 9th USENIX Conference on Operating Systems Design and Implementation, pp. 393-407, 2010.

[10] U. Zurutuza, S. Nadjm-Tehrani: "Crowdroid: Behaviour-Based Malware Detection System for Android". 1st ACM workshop on Security and privacy in smartphones and mobile devices, pp. 15-26, 2011.

[11] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck: "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket". Symposium on Network and Distributed System Security (NDSS), pp. 1-15, 2014.

[12] L.D. Coronado-de-Alba, A. Rodríguez-Mota, P.J. Escamilla-Ambrosio: "Feature Selection and Ensemble of Classifiers for Android Malware Detection". 8th IEEE Latin-American Conference on Communications (LATINCOM), pp. 1-6, 2016.

[13] J. Li, L. Zhai: "Research of Android Malware Detection Based on Network Traffic Monitoring". 9th IEEE Conference on Industrial Electronics and Applications, pp. 1739-1744, 2014.

[14] H.H. Kim, M.J. Choi: "Linux Kernel-based Feature Selection for Android Malware Detection". Asia-Pacific Network Operation and Management Symposium, pp. 1-4, 2014.

[15] R. Andriatsimandefitra, V.V. Triem Tong: "Detection and Identification of Android Malware Based on Information Flow Monitoring". IEEE 2nd International Conference on Cyber Security and Cloud Computing, pp. 200-203, 2015.

[16] S. Hou, A. Saas, L. Chen, Y. Ye: "Deep4MalDroid: A Deep Learning Framework for Android Malware Detection Based on Linux Kernel System Call Graphs". IEEE/WIC/ACM International Conference on Web Intelligence Workshops, pp. 104- 111, 2016

[17] A. Silva, J. Simmonds: "BehaviourDroid: Monitoring Android Applications". IEEE/ACM Int. Conference on Mobile Software Engineering and Systems, pp. 19-20, 2016.

[18] M.W. Afridi, T. Ali, T. Alghamdi, T. Ali, M. Yasar: "Android Application Behavioural Analysis through Intent Monitoring". 6th International Symposium on Digital Forensic and Security (ISDFS), pp. 1-8, 2018.

[19] P. Feng, J. Ma, C. Sun, X. Xu, Y. Ma: "A Novel Dynamic Android Malware Detection System with Ensemble Learning". IEEE Access, vol. 6, pp. 30996-31011, 2018.

[20] W. Li, Z. Wang, J. Cai, S. Cheng: "An Android Malware Detection Approach Using Weight-Adjusted Deep Learning". International Conference on Computing, Networking and Communications (ICNC), pp. 437-441, 2018.

[21] Z. Ni, M. Yang, Z. Ling, J. Wu, J. Luo: "Real-time Detection of Malicious Behaviour in Android Apps". International Conference on Advanced Cloud and Big Data, pp. 221-227, 2016.

[22] J.D. Koli: "RanDroid: Mandroid Malware Detection Using Random Machine Learning Classifiers". IEEE Int. Conference on Technologies for Smart-City Energy Security and Power, pp. 1-6, 2018.

[23] Y. Yao, L. Zhu, H. Wang: "Real-time Detection of Passive Backdoor Behaviours on Android System". IEEE Conference on Communications and Network Security (CCNS) - 1st International Workshop on System Security and Vulnerabiltiy (SSV), pp. 1-9, 2018.

[24] Android: "API reference". Available at https://developer.android.com/reference.

[25] M. Bokhorst: "NetGuard: A simple way to block access to the internet per application". Available at https://github.com/M66B/NetGuard/.

[26] J. Camacho, A. Pérez-Villegas, P. García-Teodoro, G. Maciá-Fernández: "PCA-based multivariate statistical network monitoring for anomaly detection". Computers & Security, vol. 59, pp. 118-137, 2016.

[27] Eybisi: "Mobile Malware Analysis: Tricks used in Anubis". Available at https://eybisi.run/Mobile-Malware-Analysis-Tricks-used-in-Anubis.