future internet

MDPI

Article

PACER: Platform for Android Malware Classification, Performance Evaluation and Threat Reporting †

Ajit Kumar [1,*], Vinti Agarwal [2], Shishir Kumar Shandilya [1], Andrii Shalaginov [2], Saket Upadhyay [1] and Bhawna Yadav [1]

[1] School of Computing Science and Engineering, VIT University Bhopal, Bhopal 466114, India; shishir.sam@gmail.com (S.K.S.); saketupadhya@gmail.com (S.U.); bhawna.yadav2018@vitbhopal.ac.in (B.Y.)

[2] Department of Information Security and Communication Technology (IIK), Norwegian University of Science and Technology, 2815 Gjøvik, Norway; vinti.agarwal3@gmail.com (V.A.); andrii.shalaginov@ntnu.no (A.S.)

* Correspondence: ajitkumar.pu@gmail.com

† This paper is an extended work from earlier work "PACE: Platform for Android Malware Classification and Performance Evaluation".

Received: 15 January 2020; Accepted: 7 April 2020; Published: 12 April 2020

Abstract: Android malware has become the topmost threat for the ubiquitous and useful Android ecosystem. Multiple solutions leveraging big data and machine-learning capabilities to detect Android malware are being constantly developed. Too often, these solutions are either limited to research output or remain isolated and incapable of reaching end users or malware researchers. An earlier work named *PACE* (Platform for Android Malware Classification and Performance Evaluation), was introduced as a unified solution to offer open and easy implementation access to several machine-learning-based Android malware detection techniques, that makes most of the research reproducible in this domain. The benefits of *PACE* are offered through three interfaces: Representational State Transfer (REST) Application Programming Interface (API), Web Interface, and Android Debug Bridge (ADB) interface. These multiple interfaces enable users with different expertise such as IT administrators, security practitioners, malware researchers, etc. to use their offered services. In this paper, we propose *PACER* (Platform for Android Malware Classification, Performance Evaluation, and Threat Reporting), which extends *PACE* by adding threat intelligence and reporting functionality for the end-user device through the ADB interface. A prototype of the proposed platform is introduced, and our vision is that it will help malware analysts and end users to tackle challenges and reduce the amount of manual work.

Keywords: android malware; machine learning; static and dynamic features; cyber threat intelligence; threat report generation; reproducible research

1. Introduction

Malware binary is a compiled "set of instructions" or a "computer program" written *intentionally* to perform malicious activities that range from simple pop-up display to unauthorized access of the device [1]. In last two decades, malicious software has been targeting almost all kinds of computing platforms. Over a decade ago, the Android platform gained the majority market share of smartphone operating systems (OS) (see Figure 1—For 2020 only January data is used while for other years an average of 12 months data is used.)and has become the de-facto OS for smartphones (https://gs.statcounter.com/os-market-share/mobile/worldwide/2015). This popularity has attracted attackers and malware writers alike and resulted in exponential growth in *Android malware* (malware specially written for Android OS as the target is called Android malware). Figure 2 depicts Android malware growth from 2012–2019. Recent security threat reports from McAfee (https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2019.pdf) and Quickheal (https://www.

quickheal.co.in/documents/threat-report/QH-Annual-Threat-Report-2019.pdf) stated not only a rise in Android malware, but also a significant rise in its complexity and scope, e.g., McAfee's highlighted that the creation of recent mobile malware involves *broader groups* than an individual attacker who is simply spying, or sending premium SMS from the victim's mobile phone.

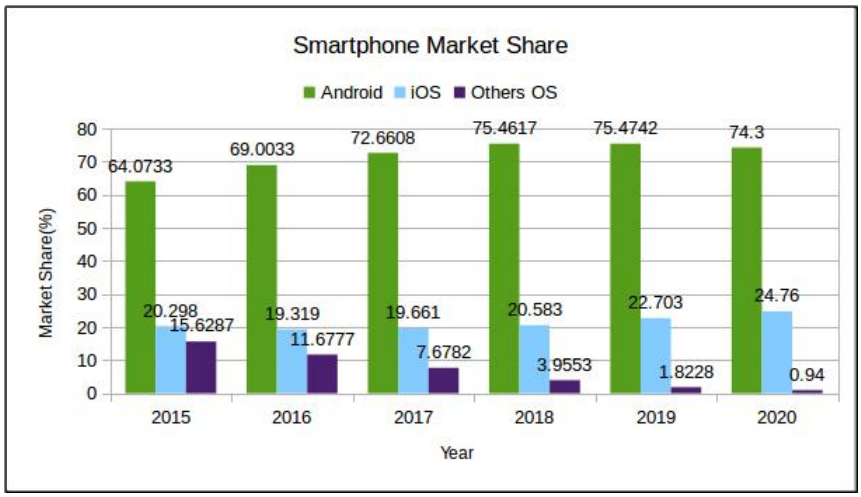

Figure 1. Smartphone Market Share 2015–20 [statcounter.com].

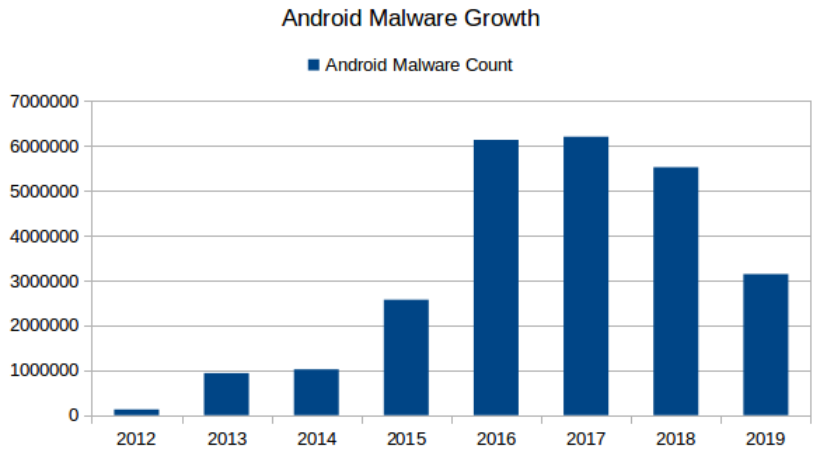

Figure 2. Android Malware Growth 2012–19 [AV-test.org].

Typically, malware detection techniques are grouped into two main classes i.e., *signature-based* and *non-signature-based* [2]. Signature-based detection determines signature, i.e., "a specially arranged sequence of bytes" by scanning a program and matching it with known signatures database to decide whether the program is malicious or not. Non-signature-based detection creates the behavioral profile of a program by executing it in a safe environment and measures its deviation from known behaviors to identify whether the program is malicious or benign. The traditional signature-based techniques have limitations of a time gap between sample detection in the wild to signature update of the end-user client, which hinders their capabilities of detecting "unknown" and "zero-day" malware [3]. These techniques also demand high computational resources that burden the end-user device. To address these limitations and bottlenecks, non-signature-based techniques have been extensively used in recent decades [4].

In addition to the above, smartphones (as well as other smart devices) pose extra challenges for malware detection techniques in terms of low computation, memory, and power availability. These resource limitations make the use of signature-based techniques very challenging for Android device security. The small and touch-sensitive smartphone screen also poses challenges (unnoticed

touch, in which, often, attackers take advantage of the small and touch-sensitive screen and hide a button from the plain sight of end users. Unaware and naive users touch (click) on the hidden button which provides a way to perform many malicious activities by attacker, reporting, etc.) for anti-malware solutions. For example, in a recent work, authors have discussed the human factors (misconfiguration of various features such as permissions, network access etc.) for Android security threats and proposed solutions to minimize human roles by analyzing the granted security at runtime and revoking them as required; many of these misconfigurations are also caused by the physical appearance of the smartphone [5].

There exists a large piece of prior work related to malware detection, and authors have proposed solutions specifically for detecting malware in Android. Kimberly et al. [6] and many recent studies [7–11] have presented an in-depth analysis of Android malware detection techniques. For example, Junyang Qiu et al. [9] reviewed the code analysis and ML-based Android malware detection techniques and listed future challenges in Android malware analysis and detection research.

The following research works have been done in the direction of providing online services and platforms to perform analysis and classification of Android malware, quite similar to what we proposed in this article. In [12], the author developed a web-based Android malware detection and classification application following client–server architecture which helps users to submit and analyze apps based on static analysis. In Table 1, a summary of the key features of various similar platforms is presented. After a thorough study of these platforms, we observe that "most of these platforms are focused only on malware static or/and dynamic feature analysis and many of them are either not functional or not updated according to a recent version of Android OS". To the best of our knowledge, no online or offline tool exists that offers ML-based Android malware detection and classification services. However, we came across a few works where implementation code and dataset are publicly available [1,13,14]. These isolated studies motivated us to propose a platform which can reproduce distributed Android malware research on one platform and make it available to users holding different responsibilities and domain expertise.

Our earlier proposed work [15] i.e., a Platform for Android malware Classification and performance Evaluation (PACE) is built upon state-of-art in Android malware detection, in particular, ML-based and non-signature-based techniques. The reproducible research is one of the main motivations of our work using research contributions from many authors and building an end-user product which serves the need for multiple stakeholders in the security community. The core idea was presented in [15] and this proposed work is a substantial improvement over *PACE*. In this proposed work (Platform for Android Malware Classification, Performance Evaluation and Threat Reporting (PACER)), we have extended the functionality of *PACE* by enhancing threat intelligence functionality and providing a proper reporting module for the end-user device. The new threat intelligence module uses *PACE* for malware scanning and also scans the device against various published Common Vulnerabilities and Exposures (CVEs) for Android platform and apps and uses app metadata to find other threat actors like *outdated apps*, apps with *negative reviews*, etc. The reporting module aims to simplify the threat and vulnerability reporting by providing an in-depth report (PDF format) on malicious apps, vulnerable apps (app versions with known CVEs), outdated apps, etc. for any Android smartphone.

The major contributions of *PACER* are:

1. Reproducible and transparent research;
2. Improved Android malware detection;
3. Multi-application integration;
4. Triaging and fast incident response;
5. Android threat intelligence;
6. Simple, descriptive and readable in-device threat report.

The performance gain and differentiated points regarding the proposed work *PACER* lies in its capabilities to provide an easy research platform for Android malware detection, generating threat intelligence and effective reporting. It must be noted that the *PACER* does not improve the performance of any individual Android malware detection techniques, but collectively it will help to design, develop, and share detection techniques to the users (researchers, end users, and product developers).

Table 1. Online Platform Related to Android Malware.

Platform	Features	Open-Source	ML-Based	Status
VirusTotal (https://www.virustotal.com/)	Scan with Commercial AV	No	No	Active
Apkdetect (https://www.apkdetect.com)	Android malware analysis and classification	No	No	Limited Access
AndroZoo (https://androzoo.uni.lu/)	Android Dataset and Analysis	No	No	Only Dataset Access
Mobile-Sandbox (http://mobilesandbox.org)	Analysis and Forensic Features extraction	Yes (https://github.com/mspreitz/mobile-sandbox)	No	Outdated
NVISO ApkScan (https://apkscan.nviso.be/)	Static and Dynamic Scanning	No	No	Going Offline (https://blog.nviso.be/2019/10/01/sunsetting-nviso-apkscan/)
JoeSandbox (https://www.joesandbox.com/#android)	Analysis and Detection	No	No	Limited Access
MobSF (https://github.com/MobSF/Mobile-Security-Framework-MobSF)	Static and Dynamic Analysis	Yes	No	Active (Local Machine)
SandDroid (http://sanddroid.xjtu.edu.cn)	Static and Dynamic Analysis	No	No	Active
AMAaaS (https://amaaas.com)	Static and Dynamic Analysis	No	No	Active
AVC UnDroid (https://undroid.av-comparatives.org/about.php)	Static Analysis	No	No	Active
PACE	Analysis and Detection	Yes	Yes	Active

The remainder of this paper is organized as follows. In Section 2, we describe the related literature and summarize research gaps. In Section 3, we describe the architecture and other components of the proposed framework. A detailed description of the original system *PACE* is presented in Section 4, upon which we build substantially improved *PACER*. The experimental setup and implementation details are elaborated in Section 5. Finally, the paper is concluded in Section 6.

2. Background

The popularity and storage capacity (personal and financial informational) of Android devices have made them a prime target for attackers. There have been many previous works which have proposed solutions to detect and prevent malware attacks on Android devices. Machine Learning (ML)-based solutions are addressing bottlenecks such as the inability to detect "unknown" and "zero-day" malware and regular signature updates of traditional signature-based detection. In 2012, Google integrated *Bouncer* (http://googlemobile.blogspot.com/2012/02/android-and-security.html) into the *Google Play Store* which scans every application and developer account before allowing them to appear in the store, and then keeps a regular check on in-device apps through *Google Play Protect* (*https://support.google.com/accounts/answer/2812853?hl=en*). This work is based on machine-learning algorithms and scanning devices in real time. Over time, it has been demonstrated that Google's *Bouncer* is not perfect at detecting malicious Android applications and specially crafted malware can evade detection (https://jon.oberheide.org/blog/?p=1004).

In the literature, ML-based Android detection is either considered to be a *binary-class* or *multi-class* classification problem. In the case of binary classification, malware and benign software are considered to be two classes. However, in multi-class classification, Android malware is grouped into various malware families [16–20]. Currently, the proposed work is focused on binary classification and does not support multi-class classification, which is mostly used for finding malware families i.e., variants of known malware. Based on the type of feature extraction method, the ML-based Android malware detection is grouped into two main categories i.e., static and dynamic detection. Static detection works with features that are extracted through static analysis, and the dynamic detection techniques are based on features that are extracted through dynamic analysis. The Section 2.1 summarizes and presents solutions proposed in the literature related to static and dynamic feature-based detection (Sections 2.1.1 and 2.1.2, respectively). The generation of a report based on the various OS and app activities on Android devices is a key area of research that attempts to fulfill different requirements such as legal requirements, security analysis, and threat intelligence through monitoring, observing, extracting, and analyzing apps to OS interactions and vice versa. Section 2.1.3 provides a brief introduction of the literature related to different types of the reporting system for Android devices.

2.1. Machine-Learning-Based Android Malware Detection

In the last decade, various ML methods have been developed for detecting and classifying Android malware. Mostly, these methods are non-signature-based and are capable of handling some of the common bottlenecks of traditional signature-based detection methods, such as the detection of "unknown" and "zero-day" malware. Based on the feature extraction method, ML-based techniques are grouped as static, dynamic and hybrid [12]. In the static method, the Android app is not executed, while the dynamic method executes samples in a controlled environment. The hybrid method is a combination of static and dynamic methods. In the subsequent Sections 2.1.1 and 2.1.2, literature works related to the static and dynamic method are presented and discussed in detail. Hybrid features are not considered in the proposed work and are normally the combination of static and dynamic methods, thus not considered in this work.

2.1.1. Static Features-Based Detection

In the static method, features are extracted from the Android application package, i.e., *.apk* without installing or executing them. The static analysis focuses mainly on the structure and static features of apps, thus the static analysis is simple and faster than the dynamic analysis. There have been many research works based on static features such as permissions, metadata, API calls, intent, and various other information obtained from the manifest file. Table 2 lists and summarizes some of the popular works based on static features and ML-based Android malware detection (False Positive Rate (FPR), True Positive Rate (TPR), Not Available (NA), Support Vector Machine (SVM)). The permissions

requested by Android applications have been the major static feature and are used in many research works to train machine-learning models [1,21–27]. Most of the earlier works have used all permissions as a feature vector (in a few of the works, feature selection is applied later) while a recent work *SigPID* [28] applies three levels of pruning to identify the most significant permissions (22 permissions) to address the scaling issues of earlier works and make scanning faster. The authors achieved 93.62% accuracy in 4–32 times less analysis time. Besides permission features, other static features such as API calls [29], intents, app metadata [21] and other values from the manifest file are used. The static features are also extracted from the source code performing reverse engineering using various tools of the Android application [27]. These static features are used either individually or in combination with other static features [30].

Table 2. Android Malware Detection Using Static Features.

Authors	Features	Performance Metrics			Dataset	Best Model
		Accuracy	FPR	TPR		
Di Cerbo et al. [21]	Permission and Metadata	NA	NA	NA	Not Available	Apriori
Sanz et al. [22]	Permission	86.41	0.19	0.91	Not Available	Random forest
Sanz et at. [31]	Manifest	94.83	0.05	0.94	Not Available	Random Forest
Ghorbanzadeh et el. [23]	Permission	0.65	0.65	0.65	Not Available	Neural Network
Yerima et al. [24]	Permission	92.1	0.06	0.90	Available	Bayesian
Peiravian et al. [29]	Permission and API	93.60	0.92	0.88	Available	Bagging
Daniel et al. [30]	Manifest and Source code	93.90	NA	NA	Available	SVM
Ajit et al. [1]	Permission	94.84	0.95	0.95	Available	Random Forest
Milosevic et al. [27]	Permission	89	NA	NA	Available	Ensemble
Milosevic et al. [27]	Source Code	95.1	NA	NA	Available	Ensemble
Jin Li et al. [28]	Permission	93.62	NA	NA	Not available	SVM

2.1.2. Dynamic Features-Based Detection

Dynamic features are extracted from the Android application package, i.e., *.apk* after installing or executing them in a controlled environment (emulator, virtual machine, etc.). The dynamic analysis focuses mainly on running time and interactive features of apps, therefore it is more complex and slower than the static analysis. The usability of dynamic analysis, in fact, lies in the ability to detect "obfuscated" and "polymorphic" malware which mostly escapes under static analysis [4]. Finding optimum running time for dynamic analysis is also difficult, as the start of malicious activities can vary from sample to sample [32]. The dynamic features are extracted through system calls, network activities, user interaction, Central Processing Unit (CPU) consumption, and other phone activities [4]. Table 3 lists and summarizes some of the popular works related to dynamic analysis using dynamic features and ML-based Android Malware detection (Input/Output (I/O), Read/Write (R/W)).

Table 3. Android Malware Detection Using Dynamic Features.

Authors	Features	Performance Metrics			Best Model
		Accuracy	FPR	TPR	
BrandonAmos et al. [33]	App traces	94.53	14.85	97.66	Random Forest
Burguera et al. [34]	User's trace (Crowdsourcing)	NA	NA	NA	Clustering
Vaibhav et al. [35]	Only Dynamic analysis	NA	NA	NA	NA
Chieh Wu et al. [32]	Network, File I/O etc.	86.1	0.857	NA	SVM
Alam et al. [36]	Features based on CPU, Memory, Battery etc.	99	NA	NA	Random Forest
Dai et al. [37]	API and corresponding arguments	NA	NA	NA	NA
Zhenlong Yuan et al. [38]	Static and Dynamic	96.5	NA	NA	Deep Learning (DBN)
DroidDetector [39]	Static and Dynamic	96.76	NA	NA	Deep Learning (DBN)
Shabtai et al. [40]	Network and Device Activities	NA	NA	NA	Anomaly-Based
Jang et al. [41]	Integrated system logs	98	NA	NA	Profiling
Chang et al. [42]	Network activities, and File R/W	97	3	97.1	Random Forest

2.1.3. Threat Intelligence Reporting

A report is an organized representation of data that helps to understand data easily and is helpful in sharing the information with others. The two main attributes of any report are that it must be for a specific audience and for a specific purpose (https://en.wikipedia.org/wiki/Report). In this regard, a technical report becomes time-consuming and tedious to write and demands a specific set of skills for both the creator and the reader. For example, presenting the security analysis report of an Android device that includes various threats posed by installed applications and the Android OS is very complex and requires high technical skills. Thus, the automatic generation of a report based on the various OS and app activities on Android devices is a key area of research that makes attempts to fulfill different requirements such as legal requirements, security analysis, and threat intelligence through monitoring, observing, extracting, and analyzing apps to OS interactions and vice versa. Kevin et al. have presented an automated crash reporting system name *CRASHSCOPE* [43]. The main task of *CRASHSCOPE* is to generate an automated augmented report after the occurrence of a crash in an Android device by any app. The report includes screenshots, crash reproduction steps, and script, and stack trace. The author Justin Grover has proposed a reporting system for forensic purposes that automatically collects system activities and provides a report for various stakeholders such as incident responders, security auditors, proactive security monitors, and forensic investigators [44]. This reporting system provides system traces as a dataset, but similar reporting can be also used for threat intelligence reporting. The *ANANAS* [45] also has a reporting module that generates a readable report by applying various filters on the stored raw data in the database. The raw data (various log files) are collected and stored in the database after performing the analysis through other modules of *ANANAS*. Moran et al. presented their work *FUSION* which helps in the automation of bug reproduction and reporting for mobile apps [46]. In this paper, we propose *PACER*, which extends *PACE* by adding threat intelligence and reporting (pdf) for the end-user device through *ADB interface*.

3. PACER: Platform for Android Malware Classification, Performance Evaluation, and Threat Reporting

After going through the aforementioned literature on ML-based Android malware detection, and studied the working of various platform related to Android malware and automated reporting systems, our observations are summarized as follows:

- There are many quality studies on Android malware detection using both static and dynamic features, but they all are up to academic research level and are not available to end users or researchers in a usable form. Our earlier work *PACE* is an attempt to fill this gap between academic research and the end user.
- The comparison of various detection techniques is complex and difficult because of varying datasets and the unavailability of implementation of the earlier works. With *PACE* we offer a common platform to compare the results of multiple methods on the same dataset and evaluate performance.
- The reproducibility and transparency is an important concern of the applied research domain. *PACE* provides a platform for conducting reproducible and transparent research.
- The implementation of many earlier studies and datasets are either closed source or available on request. As research work grows older, keeping pace with contact and project becomes difficult, as the research contact (i.e., website, link, email) became obsolete.
- Very often, research works are not available in a usable format, with various interface options such as REST API, Web Interface, ADB, etc. *PACE* is a solution that makes research available to different kinds of users i.e., from end users to security practitioners.
- With a centralized submission system, *PACE* would be very helpful for Android threat intelligence by analyzing the sample daily and sharing the intelligence through the dashboard.
- Automatically generating a simple, meaningful, and readable threat intelligence report for the user device is very important for convincing and making the naive and non-technical user understand about security risks. With the extended work *PACER*, we try to address the issue of threat accessibility and understandability for the naive user by providing a simple, meaningful and readable report that has *PACE* scanning results along with other threat results such as *CVE* matching, outdated apps, etc.

The proposed work *PACER* is an extension of our previous work *PACE*. It extends the functionality of *PACE* and provides mechanisms to generate an automated threat intelligence report of an end-user device. The Threat Intelligence Reporting (TIR) module can be developed in two ways either as a plugin for existing *PACE* (can use ADB interface to access device) or as an independent system, and can access *PACE* scanning through an API interface. With such open architecture, it is possible to include other malware classification tools (which have interfaces to accept apk as input and provides classification results as 'malware' or 'benign') into *PACER* along with *PACE* or by replacing it.

Currently, we have developed TIR as *PACER*, which is an independent system and uses *PACE* for malware scanning results. Figure 3 depicts the architecture of *PACER*, which has *PACE* as a module. *PACER* is mainly divided into three modules: *Data Processing*, *Threat Analyzer*, and *Report Generator*. Each of these modules are explained in detail in the following sections.

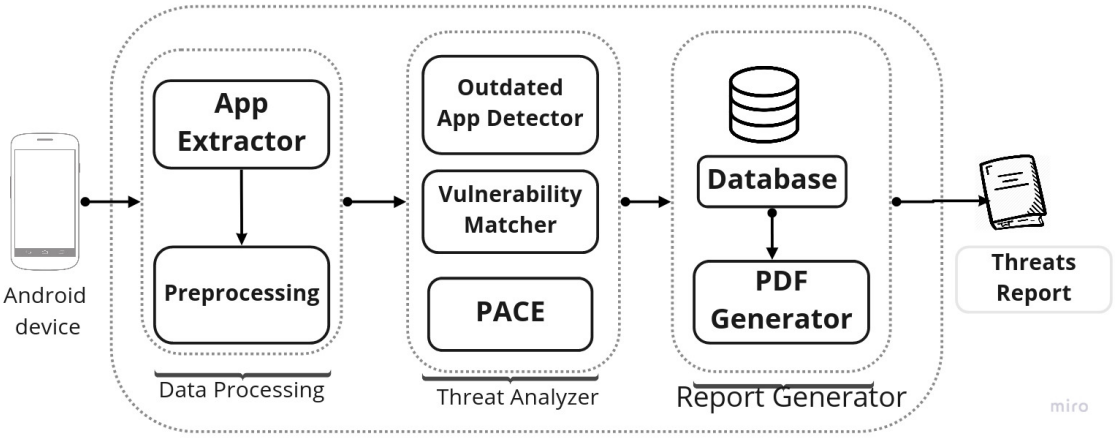

Figure 3. Block Representation of PACER.

3.1. Data Processing

Data processing has two main components: *App Extractor* and *Pre-Processing*. The *App Extractor* component uses the ADB interface to extract all the installed apps on the attached smartphone. It is very similar to the ADB interface of *PACE*, so adding a reporting module to *PACE* would be very simple, but currently *PACER* is implemented as a separate tool for simplicity and ease of use. The *Pre-Processing* component performs functions like extracting app name, version, etc. which can be used in the *Threat Analyzer* module for vulnerability matching, outdated app detection, or getting a malware classification result from *PACE*.

3.2. Threat Analyzer

The *Threat Analyzer* module is divided into three components: *Outdated App Detector*, *Vulnerability Matcher*, and *PACE*. Each of these components provides threat intelligence such as threats due to outdated, vulnerable, and malicious apps, respectively.

3.2.1. Outdated App Detector

An outdated app poses a serious threat for a smartphone because there are possibilities of existing known vulnerabilities in the outdated version and attackers are often well versed with attacking methods and target devices with such apps. The *Outdated App Detector* components takes the app name and version as input and matches them with gathered information (metadata from app stores such Google Play etc.). The output (such as whether app is outdated or not, and new version of app) of *Outdated App Detector* is stored in a database that is used for report generation. This component is important because many times only a popular app gets attacker attention and hence has a known CVE number, while other apps, despite vulnerabilities, do not have a popular CVE number or a publicly available patch. Hence such outdated apps pose a security risk in the case of a targeted attack because only the original developer can make the patch available, through updated versions.

3.2.2. Vulnerability Matcher

Software vulnerabilities also known as *bugs* are coding errors that cause the system to make an unwanted action [47]. The Common Vulnerabilities and Exposures (CVE) is the code prefix with a unique number, which all known vulnerabilities get assigned. Each of these vulnerabilities are reported by security practitioners and have a unique number. For most of these CVEs, proof-of-concept (PoC) exploits are publicly available, so devices with app version with published CVEs are an easy target [48,49]. The purpose of ethical disclosure and publication of vulnerability is to force the maintainer of an application to release a patch for each CVE. Due to the lack of information and knowledge, often naive users do not update their vulnerable apps. The *Vulnerability Matcher* component

will match all the installed apps with known CVE databases and highlight the vulnerable apps in the report. The vulnerability matching is a one-to-many problem, i.e., one app can have more than one known CVE, so the *Vulnerability Matcher* must find all the CVEs and include them in the report. As the CVE database contains all the reported vulnerabilities of the applications, so over time there are many CVEs. By December 2019, CVE Database contained 5725 Android vulnerabilities. Thus, each installed app on the device must search among these 5725 Android vulnerabilities, which increases the space and time complexity. To reduce searching complexity, the best of three string-search algorithms—Boyer–Moore string-search, Knuth–Morris–Pratt and Naive string-search algorithm—is used. Table 4 shows that space requirement and time required for pre-processing and string match of each algorithm.

Table 4. Comparison of String-Matching Algorithms.

Algorithms	Pre-Processing	Matching	Space
Naive String-Search	None	$\theta(nm)$	None
Knuth–Morris–Pratt	$\theta(m)$	$\theta(n)$	$\theta(m)$
Boyer–Moore	$\theta(m+k)$	$\mathcal{O}(mn)$	$\theta(k)$

Boyer–Moore String algorithm is well suited for applications in which the pattern is much shorter than the text or where it persists across multiple searches. The Boyer–Moore algorithm uses information gathered during the pre-processing step to skip sections of the text, resulting in a lower constant factor than many other string-search algorithms. Thus, the best result is shown by Boyer–Moore string algorithm, and *PACER* uses this for CVE matching.

3.2.3. PACE: Platform for Android Malware Classification and Performance Evaluation

PACE is an online platform for the analysis and classification of Android malware using machine-learning techniques. The overall workflow of *PACE* is divided into four components: (1) *Model Building* (2) *PACE Architecture* (3) *Interface* and (4) *Target Users*. For content clarity and maintaining the reading flow, a detailed discussion of various components of PACE is presented further in Section 4.

3.3. Report Generator

The *Report Generator* has a database and *PDF generator* components. The PDF generator takes values from the database which has output of threat analyzer modules. Currently, the *Report Generator* module provides a report of all installed apps upon a three-threat vector which includes outdated apps, vulnerable apps, and malicious apps. The report of each app also includes suggestions such as the app's need to update or be removed, or if a new version is available. This threat vector can be increased by adding more components to the threat analyzer module. This way, the customization of the generated report is easy, and researcher can add/remove information into the report by adding new modules for a new threat vector, or can restrict the information by applying a filter into a report generator module. It is also worth mentioning that currently the *PACER* is only able to generate a report in the *PDF* format, but as the analysis results are stored in a database, that can be exported to other useful formats such as JSON and XML by developing suitable modules for respective formats. The sample report of two devices are added into Supplementary Materials.

4. PACE: Platform for Android Malware Classification and Performance Evaluation

PACE provides malware classification results to PACER's for creating simple, useful, and readable reports. This section provides an in-depth explanation of all four components of *PACE*, namely *Model Building*, *PACE Architecture*, *Interface*, and *Target Users*. Model Building is an offline process and under this, various machine-learning models are trained. The *PACE* architecture is the core part of the workflow and it offers analysis and classification services to users via multiple interfaces,

i.e., REST API, Web Interface, and ADB. A user can interact with *PACE* by using any of these interfaces. The different interface enables users with different expertise (such as non-technical users, IT administrators, and security practitioners and malware researchers) to use offered services as per demand. Figure 4 depicts the overall workflow of *PACE* and the interaction among different components. In the following subsections, all four components of *PACE* are described in detail.

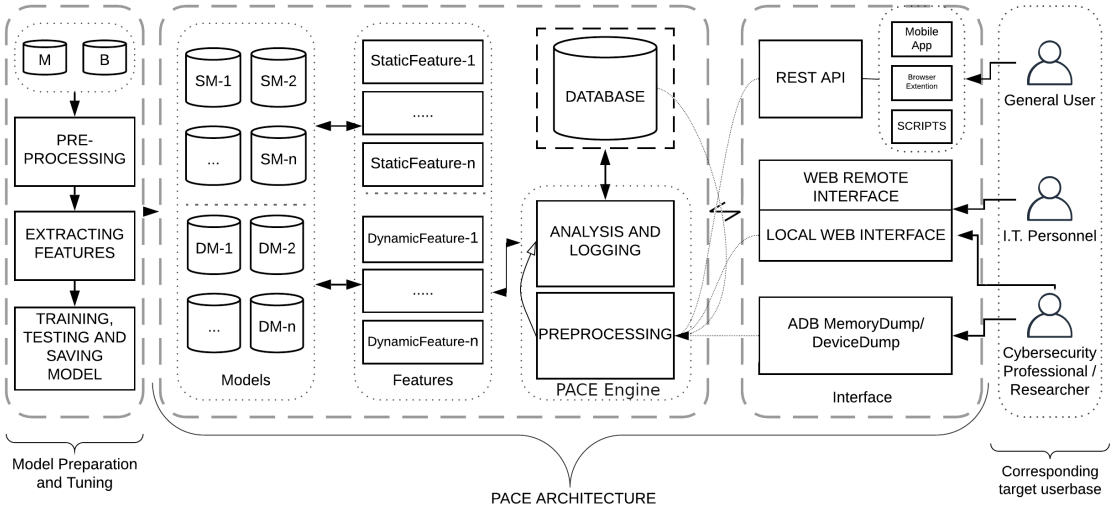

Figure 4. Workflow and Architecture of PACE. [M-Malware; B-Benign; SM-Static Model; DM-Dynamic Model].

4.1. Model Building

Model Building is a compulsory part of the platform. It performs all steps of the machine-learning process such as data collection, pre-processing, feature extraction, training, testing, and saving the trained and tested model. Model Building is an offline process, where models are trained separately from the running instance of *PACE* and are added incrementally to *PACE* on a completion basis. The selection and addition of a model depend upon the author's permission and legal boundary to reproduce the selected research works. Currently, for testing and demo purposes, we have trained two models based on Random Forest and Support Vector Machine. These models are based on static features (permissions) and have accuracy of 68% and 93% without any feature engineering or parameter tuning of algorithms, respectively. In Figure 4, the first vertical layer depicts the Model-Building process from dataset collection to the saved trained model.

The Figure 5 depicts the output of both trained models; in similar ways, the result of other models will be displayed after they are added to the *PACE*.

ML Engine Results

No.	Name	Classification [Confidence]	Feature Type	Base Operation	Author
1	Random Forest (88%)	MALWARE [67.60000000000001]	Permissions	Random Forest	Bhawna Yadav
2	SVM (83%)	MALWARE [92.80000000000001]	Permissions	Support Vector Machine	Saket Upadhyay

Figure 5. Output of Trained models.

The dataset used for training models is the same and collected from AndroZoo project (https://androzoo.uni.lu/). During the pre-processing phase, depending on the features

a model is using, several operations are performed on each sample from the collected dataset. These are given below:

1. File Type Detection: As a first step, it is very important to detect the file type of each sample since further steps are file type-dependent.
2. Duplicate Removal: The same sample in the dataset can be present with a different file name. To have only a unique sample in the dataset, file hash is used to filter duplicate samples.
3. Class Labeling: Since *PACE* is used to perform binary classification, the dataset should have samples from both classes. Class labels are decided based on VirusTotal result. We considered a sample as benign if the VirusTotal's engine shows that the sample is clean, otherwise the sample is considered to be malware.
4. Sample Reversal: Each sample is reversed as per the demand of the feature type of a model. For example, if the underlying model is using *permissions-based* features for training, the manifest file will be extracted from each sample by simply unzipping the .apk file. On the other hand, if the model is dependent upon features extracted from the source code, the sample would be reverse engineered using tools like apktool, etc. Likewise, various methods will be applied to extract respective features.

The pre-processed dataset will be passed to the next stage for feature extraction, and depending upon the required feature, the respective feature extractor module will be executed on the dataset. For example, to build a permission feature set, a permissions extractor will be executed on the pre-processed dataset and the result in the set structure will be saved with proper class label, i.e., for malicious application CLASS TYPE 1 and for benign CLASS TYPE 0. After processing all the samples and converting features in the required format, the feature set would be input to the machine-learning algorithm, which is referenced from respective research papers to create a malware classifier. Once the training and testing of the model are completed as per the description given in respective research work, then the tested model would be saved (making model persistence) to be used in the *PACE*. The model building is a one-time process and will be done for all selected research work. The end user or the interface of *PACE* has no direct connection with this module and can only access if the model is added to the *PACE*.

In Section 4.2 the PACE architecture is explained in detail and the subsequent sections explain the *interface* and *user type* in detail.

4.2. PACE: Architecture

The core of *PACE* workflow is the *PACE* architecture, which comprises *trained models*, *extracted features*, *analysis*, and *pre-processing*. The *PACE* architecture offers its services via an *interface* layer, and a user can only interact with *PACE* via the defined interface's options. In "Figure 4" the central part depicts the main *PACE* architecture which is embedded between the Model-Building and Interface layers.

4.2.1. Models

The first component of the *PACE* architecture is the *trained models* which is the collection of all trained models added to or set to be added to *PACE*; the process is explained in Section 4.1. The Static Model (SM) is a model trained with static features and the Dynamic Model (DM) is a model trained with dynamic features. All the models that are trained and tested are saved in the directories and integrated into *PACE*. The *PACEngine* is the module that is used to integrate the various models into *PACE*. The *PACEngine* submits the feature set to the models as per their configuration and receives the prediction and confidence calls from each model, and passes the result to the interface layer. The *PACEngine* allows maintenance of the models in a systematic way and makes the architecture extensible, i.e., adding a new model becomes very simple.

4.2.2. Feature Set

The feature set is the collection of similar types of features, e.g., for a permission feature set all the features will the extracted permissions. As discussed earlier, there are two ways of extracting features from Android, i.e., static and dynamic, and there are different types of static and dynamic features. Under static feature permissions, metadata, intents, API calls, and source code are mostly used and dynamic features are network activities, activities related to File I/O, Memory R/W, and CPU and battery usage, Inter Process Communication (IPC), and system call. This difference between static and dynamic is represented as Static Feature 1-n and the Dynamic Feature 1-n in the Figure 4. The type of feature depends upon the selected research works present in *PACE*, and so features are extracted and processed accordingly. The user can select the models for scanning their sample and so, according to the selected models, the features will be extracted, or features for all available models will be extracted.

4.2.3. Pre-Processing

This pre-processing module is similar to the Model-Building module, except this pre-processing is performed online. The submitted Android app under inspection is passed to the processing module, which will then prepare the sample for feature extraction by performing various processing such as the removal of any irrelevant information, converting the app into a different format, i.e., .dex, .smali, etc. During the pre-processing, the submitted app reversed code is also packed in a zip file and attached to the main interface so that the user can download the source code of the submitted application and use that further in the analysis. The data generated during the pre-processing along with the predicted classification label is also saved in the database, which can later be used for re-training and testing. The saved result can act as a cache and can be used to fasten the response by not performing all steps repeatedly for known app submission. The pre-processing module has also passed information to the analysis and logging module. The analysis and logging module is explained in further Section 4.2.4.

4.2.4. Analysis and Logging

The analysis and logging module is an important module and provides useful insights about the submitted application and shows the correlation pattern with malicious and benign class. The output of the analysis and logging module serves as threat intelligence and helps to understand the trend of Android malware, e.g., listing the permissions and showing the permission trend helps to understand the kind of attacks. This module is extensible and different analysis engines can be added over time. Currently, it shows permissions and entropy. Later dynamic analysis results such as CPU, battery and memory usage, system call graph, etc. will be added to the *PACE* panel.

4.3. PACE: Interface

The *PACE* is designed to keep it extensible, so it has adopted a layered approach. The interface is separate from the core PACE architecture, which makes it easy to add and remove new ways of access to the *PACE*. In Figure 4, the interface is at the right end of the workflow and different types of users interact with *PACE* through the *Interface* layer. The services of *PACE* can be accessed by a set of methods that are defined in the *interface* module. Such separation helps to choose the method according to need and offers the option to integrate *PACE* services into other services seamlessly. The three main methods under the *Interface* module are *REST API*, *Web Interface* and *ADB Interface*.

4.3.1. REST API

The **REST API** makes it easy to offer *PACE* services via various applications such as *Android Application* and *Browser Extension*, and as *Scripts*. For example, the scripting method makes it easy to integrate *PACE* into the Android app store, which is very important, as much Android malware is pushed to the user device via a third-party app store. *Google Play Protect* is Google's in-built anti-malware solution which uses an ML-based engine to scan apps in real time (https://www.

android.com/intl/en_in/play-protect/). Even then, many times, malicious Android applications have evaded detection and stayed in the Play Store for a long time (https://thenextweb.com/apps/2019/10/01/google-play-android-malware-2/). The current implementation of the proposed work *PACER* uses REST API to get malware classification results for all installed apps of the device and include them in the threat intelligence report.

4.3.2. Web Interface

The web interface can be accessed by running *PACE* on a *Local Machine* individually or accessed the offering of *PACE* through our hosted *Remote Machine*. The local interface differs slightly as it is targeting security practitioners and researchers. The local interface provides options to pass the path of the local directory as input to *PACE* and the result will return in desired output structured formats (JavaScript Object Notation (JSON), Comma Separated Values (CSV), and Extensible Markup Language (XML)). This helps to perform research and test many samples which will not be possible with the online version. The online web interface only allows the submission of one sample at a time and the provision of a result for the same.

4.3.3. ADB and Memory Dump

The ADB is a client–server program used in Android application development. The Android Debug Bridge is part of the Android Software Development Kit (SDK) and is made up of three components: a client, a daemon, and a server. It is used to manage either an emulator instance or an actual Android device. The ADB method of *PACE interface* is also required to run *PACE* as a local instance on the same machine on which the Android device is connected or the emulator is running. Thus, the application installed on the device can be scanned with *PACE* models. This option is similar to our previous work FAMOUS [1], in which the Android device was scanned with the best performing model, which was trained with weighted permissions. This method is especially for forensic and security analysts and helps them to triage the applications and help them to prioritize the app for manual analysis and make the scanning process faster. The *ADB Connection* is used by the current implementation of the proposed work *PACER* for extracting apps from a device and submitting them to *PACE* for malware classification results.

4.4. Targeted Users

The targeted users are divided into three main groups i.e., General User, IT Personnel and Cybersecurity Professional, and Android malware researcher. Each group of users can be served by different methods available in the interface module. The grouping of users is purely based on ease of use and their preferred method; for example, the general user can access *PACE services* via the online web interface or through the Android application, which is very simple, and user will be comfortable, while on the other hand, the researcher can use a local web interface or ADB-based method to speed up their research work by bulk analysis and scanning.

5. Experimental Setup and Implementation

The proposed work *PACER* is developed as a tool (written in Python) and run on Windows OS. The earlier work *PACE* is experimented and implemented locally and an online instance is also available with limited features. This section provides details about the dataset and implementation of *PACER* and *PACE*.

5.1. Dataset

During the literature review, it was observed that most researchers use their own datasets, and very few of them are available publicly, which makes comparing detection techniques difficult. It was also observed that some of the datasets are now obsolete and research based on that dataset

cannot be reproduced hence cannot be compared with the new method. Keeping these challenges in mind, we are going to keep a common dataset for all models. The sample for current working models was downloaded from AndroZoo [50] project, which has a large collection of *benign* and *malware* Android applications (The API key was requested and obtained from AndroZoo maintainer and we are following all the given instructions for dataset use.).

As per the AndroZoo guidelines, we are not free to share the sample, but the pre-processing and feature extraction scripts will be open source and available for public usage, which will enable us to reproduce various research.

5.2. Implementation

The *PACER* is implemented as a set of Python scripts on a Windows 10 Home (64 bits OS) running with Intel (R) Core (TM) i5-7200U processor and 4 GB RAM and tested with different smartphones with Android, and threat reports are generated as a PDF file for each device (Supplementary Files represent two sample reports of two different devices). Each of the components of *PACER* are implemented with the help of various Python modules like *reportlab*, *subprocess*, *regex*, etc. The *PACE* is implemented using a Python stack i.e., using various modules and frameworks of Python. For example, *Scikit-learn* [51] is used for machine-learning models and Flask is used to design a web interface. The current version of *PACE* is being developed on a computer with basic configuration and minimum memory requirement, i.e., 8 GB RAM and 60 GB hard disk (HDD, 5200 RPM) with *i7* 8^{th} *generation* Intel processor with 8 cores. Figure 6 presents the flow of various actions and the interaction of various components of the *PACER*. Each of these tasks was implemented as a Python script and was integrated with each other within *PACER*.

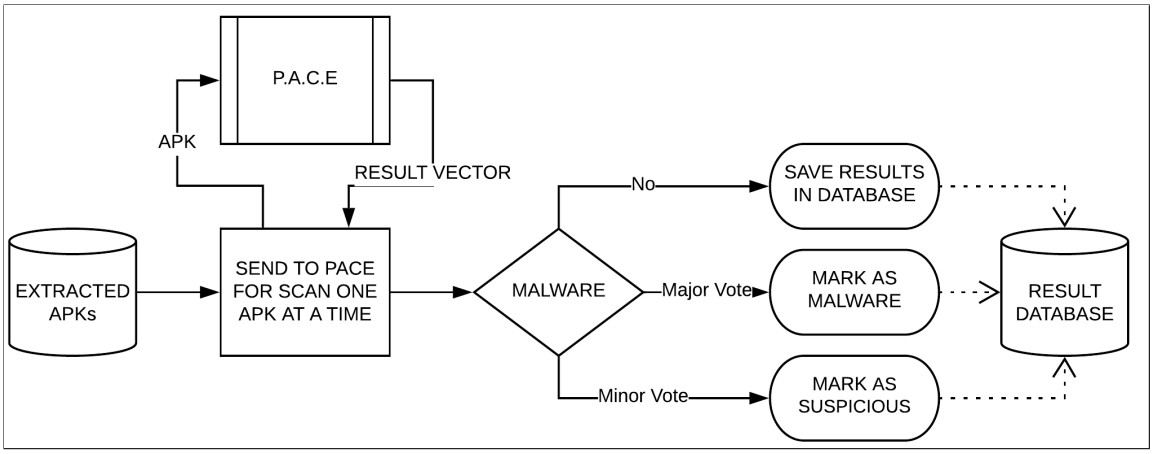

Figure 6. Flowchart of PACER.

6. Discussion and Conclusions

The proposed work *PACER* is an attempt to broaden Android malware research by making earlier research reproducible, transparent, and accessible to targeted users. Although through *PACE* interface it will be easy to get the various analysis results and compare with different detection methods, *PACER* will help to make these detection results along with other threats result available to naive users in a simple, useful, and readable format. The main limitation of the *PACE* is the author's confirmation and the legal scope. Research work can be added to the *PACE* only after getting due permission from the author that is possible within the legal scope of the work. This will make the growth of *PACE* slow but over time more authors will participate and will enrich and strengthen *PACE*. The proposed report format of *PACER* is limited in number of threat types, but can be extended by adding more scripts as the system is open, and so new scripts can be added easily.

The proposed work *PACER* is novel in its purpose and application. In this work, only limited use cases of *PACER* and *PACE* are listed and described but due to its extensible and open architecture,

there would be various other usages and accordingly required functions that can be added to both platforms. The *PACE* implementation will help to improve research in Android malware detection by making it reproducible, transparent, and simple, and the *PACER* reporting mechanism makes this research available to naive users by taking off the complex technical requirements. The *PACE* will help to improve the anti-malware applications for the Android platform by making many classification engines available via REST API, which can be easily integrated with the products. The *PACER* will also help to strengthen the Android security awareness by reporting and educating users about vulnerable and outdated apps along with other threats inputs. The *PACER* and *PACE* both will serve as supportive tools for cybersecurity and forensic practitioners by helping them to scan the end-user devices or memory dump of seized devices for prioritizing apps for in-depth analysis or finding the rouge application. The paper consists of the following major contributions:

a. A unique framework to standardize the existing studies for comparative analysis. This certainly reduces the programming efforts and code duplicity while providing a facility to compare, create, verify, and deploy new malware detection techniques.

b. Its availability in public domain under GNU General Public License allows it to run, study, share, and modify the techniques. Also, due to regular additions of newly published methods, it will be easy and quick to deliver those techniques to the end user via various available interfaces (API, App, etc.).

c. The proposed work will assist malware analysts in triaging the new sample and help to filter out samples quickly, which requires manual analysis.

d. Due to multiple machine-learning-based methods, the detection will be more accurate and the threat reporting and sharing (along with scanning result, Exif information, Permissions Ranks and a nicely Decompiled Source Code of the submitted sample ready to download, all in one place) will be fast.

For example, recently, we used a local version of *PACE* to analyze a recent piece of Android malware that was targeting Indian Android users, scamming them under the promise of offering Jio Prime Membership and 25 GB of internet daily (full report is available at https://github.com/Saket-Upadhyay/MalwareReports).

Supplementary Materials: The following are available online at http://www.mdpi.com/1999-5903/12/4/66/s1.

Author Contributions: Conceptualization, A.K. and S.U.; Data curation, B.Y.; Funding acquisition, V.A.; Investigation, B.Y. and S.U.; Methodology, A.K.; Project administration, A.S.; Software, B.Y.; Supervision, S.K.S.; Visualization, A.K.; Writing—original draft, A.K.; Writing—review & editing, V.A., S.K.S., A.S. and S.U. All authors have read and agreed to the published version of the manuscript.

Funding: This research is sponsored by the Department of Information Security and Communication Technology (IIK), NTNU Norway.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kumar, A.; Kuppusamy, K.; Aghila, G. FAMOUS: Forensic Analysis of MObile devices Using Scoring of application permissions. *Future Gener. Comput. Syst.* **2018**, *83*, 158–172. [CrossRef]
2. Suarez-Tangil, G.; Tapiador, J.E.; Peris-Lopez, P.; Ribagorda, A. Evolution, detection and analysis of malware for smart devices. *IEEE Commun. Surv. Tutor.* **2013**, *16*, 961–987. [CrossRef]
3. Faruki, P.; Ganmoor, V.; Laxmi, V.; Gaur, M.S.; Bharmal, A. AndroSimilar: Robust statistical feature signature for Android malware detection. In Proceedings of the 6th International Conference on Security of Information and Networks, Aksaray, Turkey, 26–28 November 2013; pp. 152–159.
4. Qamar, A.; Karim, A.; Chang, V. Mobile malware attacks: Review, taxonomy & future directions. *Future Gener. Comput. Syst.* **2019**, *97*, 887–909.
5. Gupta, S.; Buriro, A.; Crispo, B. A Risk-Driven Model to Minimize the Effects of Human Factors on Smart Devices. In Proceedings of the International Workshop on Emerging Technologies for Authorization and Authentication, Luxembourg City, Luxembourg, 27 September 2019, doi:10.13140/RG.2.2.22525.92649. [CrossRef]

6. Tam, K.; Feizollah, A.; Anuar, N.B.; Salleh, R.; Cavallaro, L. The evolution of android malware and android analysis techniques. *ACM Comput. Surv. (CSUR)* **2017**, *49*, 76. [CrossRef]
7. Yan, P.; Yan, Z. A survey on dynamic mobile malware detection. *Softw. Qual. J.* **2018**, *26*, 891–919. [CrossRef]
8. Felt, A.P.; Finifter, M.; Chin, E.; Hanna, S.; Wagner, D. A survey of mobile malware in the wild. In Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, Chicago, IL, USA, 17 October 2011; pp. 3–14.
9. Qiu, J.; Nepal, S.; Luo, W.; Pan, L.; Tai, Y.; Zhang, J.; Xiang, Y. Data-Driven Android Malware Intelligence: A Survey. In Proceedings of the International Conference on Machine Learning for Cyber Security, Xi'an, China, 19–22 September 2019; pp. 183–202.
10. Samra, A.A.A.; Qunoo, H.N.; Al-Rubaie, F.; El-Talli, H. A survey of Static Android Malware Detection Techniques. In Proceedings of the 2019 IEEE 7th Palestinian International Conference on Electrical and Computer Engineering (PICECE), Gaza, Palestine, 26–27 March 2019; pp. 1–6.
11. Sahay, S.K.; Sharma, A. A Survey on the Detection of Android Malicious Apps. In *Advances in Computer Communication and Computational Sciences*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 437–446.
12. Doğru, İ.; KİRAZ, Ö. Web-based android malicious software detection and classification system. *Appl. Sci.* **2018**, *8*, 1622. [CrossRef]
13. Au, K.W.Y.; Zhou, Y.F.; Huang, Z.; Lie, D. Pscout: Analyzing the android permission specification. In Proceedings of the 2012 ACM conference on Computer and communications security, Raleigh, NC, USA, 16–18 October 2012; pp. 217–228.
14. McLaughlin, N.; Martinez del Rincon, J.; Kang, B.; Yerima, S.; Miller, P.; Sezer, S.; Safaei, Y.; Trickel, E.; Zhao, Z.; Doupé, A.; et al. Deep Android Malware Detection. In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, Scottsdale, AZ, USA, 22–24 March 2017; pp. 301–308. doi:10.1145/3029806.3029823. [CrossRef]
15. Kumar, A.; Agarwal, V.; Shandilya, S.K.; Shalaginov, A.; Upadhyay, S.; Yadav, B. PACE: Platform for Android Malware Classification and Performance Evaluation. In Proceedings of the 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, 9–12 December 2019; pp. 4280–4288.
16. Zhang, L.; Thing, V.L.; Cheng, Y. A scalable and extensible framework for android malware detection and family attribution. *Comput. Secur.* **2019**, *80*, 120–133. [CrossRef]
17. Kim, H.M.; Song, H.M.; Seo, J.W.; Kim, H.K. Andro-Simnet: Android Malware Family Classification using Social Network Analysis. In Proceedings of the 2018 16th Annual Conference on Privacy, Security and Trust (PST), Belfast, UK, 28–30 August 2018. doi:10.1109/pst.2018.8514216. [CrossRef]
18. Le Thanh, H. Analysis of malware families on android mobiles: detection characteristics recognizable by ordinary phone users and how to fix it. *J. Inf. Secur.* **2013**, *4*, 213–224. [CrossRef]
19. Xie, N.; Wang, X.; Wang, W.; Liu, J. Fingerprinting Android malware families. *Front. Comput. Sci.* **2019**, *13*, 637–646. [CrossRef]
20. Massarelli, L.; Aniello, L.; Ciccotelli, C.; Querzoni, L.; Ucci, D.; Baldoni, R. Android malware family classification based on resource consumption over time. In Proceedings of the 2017 12th International Conference on Malicious and Unwanted Software (MALWARE), Fajardo, PR, USA, 11–14 October 2017; pp. 31–38.
21. Di Cerbo, F.; Girardello, A.; Michahelles, F.; Voronkova, S. Detection of malicious applications on android os. In *Computational Forensics*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 138–149.
22. Sanz, B.; Santos, I.; Laorden, C.; Ugarte-Pedrero, X.; Bringas, P.G.; Álvarez, G. Puma: Permission usage to detect malware in android. In *International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 289–298.
23. Ghorbanzadeh, M.; Chen, Y.; Ma, Z.; Clancy, T.C.; McGwier, R. A neural network approach to category validation of android applications. In Proceedings of the 2013 International Conference on Computing, Networking and Communications (ICNC), San Diego, CA, USA, 28–31 January 2013; pp. 740–744.
24. Yerima, S.Y.; Sezer, S.; McWilliams, G.; Muttik, I. A new android malware detection approach using bayesian classification. In Proceedings of the 2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA), Barcelona, Spain, 25–28 March 2013; pp. 121–128.
25. Talha, K.A.; Alper, D.I.; Aydin, C. APK Auditor: Permission-based Android malware detection system. *Digit. Investig.* **2015**, *13*, 1–14. [CrossRef]

26. Geneiatakis, D.; Fovino, I.N.; Kounelis, I.; Stirparo, P. A Permission verification approach for android mobile applications. *Comput. Secur.* **2015**, *49*, 192–205. [CrossRef]
27. Milosevic, N.; Dehghantanha, A.; Choo, K.K.R. Machine learning aided Android malware classification. *Comput. Electr. Eng.* **2017**, *61*, 266–274. [CrossRef]
28. Li, J.; Sun, L.; Yan, Q.; Li, Z.; Srisa-an, W.; Ye, H. Significant permission identification for machine-learning-based android malware detection. *IEEE Trans. Ind. Inform.* **2018**, *14*, 3216–3225. [CrossRef]
29. Peiravian, N.; Zhu, X. Machine learning for android malware detection using permission and api calls. In Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, 4–6 November 2013; pp. 300–305.
30. Arp, D.; Spreitzenbarth, M.; Hubner, M.; Gascon, H.; Rieck, K. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 23–26 February 2014.
31. Sanz, B.; Santos, I.; Laorden, C.; Ugarte-Pedrero, X.; Nieves, J.; Bringas, P.G.; Álvarez Marañón, G. MAMA: Manifest analysis for malware detection in android. *Cybern. Syst.* **2013**, *44*, 469–488. [CrossRef]
32. Wu, W.C.; Hung, S.H. DroidDolphin: A dynamic Android malware detection framework using big data and machine learning. In Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems, Towson, MD, USA, 5–8 October 2014; pp. 247–252.
33. Amos, B.; Turner, H.; White, J. Applying machine learning classifiers to dynamic android malware detection at scale. In Proceedings of the 2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC), Sardinia, Italy, 1–5 July 2013; pp. 1666–1671.
34. Burguera, I.; Zurutuza, U.; Nadjm-Tehrani, S. Crowdroid: Behavior-based malware detection system for android. In Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, Chicago, IL, USA, 17 October 2011; pp. 15–26.
35. Rastogi, V.; Chen, Y.; Enck, W. AppsPlayground: Automatic security analysis of smartphone applications. In Proceedings of the third ACM conference on Data and application security and privacy, San Antonio, TX, USA, 18–20 February 2013; pp. 209–220.
36. Alam, M.S.; Vuong, S.T. Random forest classification for detecting android malware. In Proceedings of the 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing, Beijing, China, 20–23 August 2013; pp. 663–669.
37. Dai, S.; Wei, T.; Zou, W. DroidLogger: Reveal suspicious behavior of Android applications via instrumentation. In Proceedings of the 2012 7th International Conference on Computing and Convergence Technology (ICCCT), Seoul, Korea, 3–5 December 2012; pp. 550–555.
38. Yuan, Z.; Lu, Y.; Wang, Z.; Xue, Y. Droid-sec: Deep learning in android malware detection. In Proceedings of the ACM SIGCOMM Computer Communication Review, Chicago, IL, USA, 17–22 August 2014; pp. 371–372.
39. Yuan, Z.; Lu, Y.; Xue, Y. Droiddetector: android malware characterization and detection using deep learning. *Tsinghua Sci. Technol.* **2016**, *21*, 114–123. [CrossRef]
40. Shabtai, A.; Tenenboim-Chekina, L.; Mimran, D.; Rokach, L.; Shapira, B.; Elovici, Y. Mobile malware detection through analysis of deviations in application network behavior. *Comput. Secur.* **2014**, *43*, 1–18. [CrossRef]
41. Jang, J.W.; Yun, J.; Mohaisen, A.; Woo, J.; Kim, H.K. Detecting and classifying method based on similarity matching of Android malware behavior with profile. *SpringerPlus* **2016**, *5*, 273. [CrossRef] [PubMed]
42. Chang, W.L.; Sun, H.M.; Wu, W. An Android Behavior-Based Malware Detection Method using Machine Learning. In Proceedings of the 2016 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC), Hong Kong, China, 5–8 August 2016; pp. 1–4.
43. Moran, K.; Linares-Vásquez, M.; Bernal-Cárdenas, C.; Vendome, C.; Poshyvanyk, D. Automatically discovering, reporting and reproducing android application crashes. In Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), Chicago, IL, USA, 11–15 April 2016; pp. 33–44.
44. Grover, J. Android forensics: Automated data collection and reporting from a mobile device. *Digit. Investig.* **2013**, *10*, S12–S20. [CrossRef]
45. Eder, T.; Rodler, M.; Vymazal, D.; Zeilinger, M. Ananas-a framework for analyzing android applications. In Proceedings of the 2013 International Conference on Availability, Reliability and Security, Regensburg, Germany, 2–6 September 2013; pp. 711–719.

46. Moran, K.; Linares-Vásquez, M.; Bernal-Cárdenas, C.; Poshyvanyk, D. Auto-completing bug reports for android applications. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 30 August–4 September 2015; pp. 673–686.
47. Winkler, I.; Gomes, A.T. *Advanced Persistent Security: A Cyberwarfare Approach to Implementing Adaptive Enterprise Protection, Detection, and Reaction Strategies*; Syngress: Rockland, MA, USA, 2016.
48. Mitra, J.; Ranganath, V.P. Ghera: A repository of android app vulnerability benchmarks. In Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, Toronto, ON, Canada, 8 November 2017; pp. 43–52.
49. Zhang, M.; Yin, H. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. Available online: http://lilicoding.github.io/SA3Repo/papers/2014_zhang2014appsealer.pdf (accessed on 8 April 2020).
50. Allix, K.; Bissyandé, T.F.; Klein, J.; Le Traon, Y. AndroZoo: Collecting Millions of Android Apps for the Research Community. In Proceedings of the 13th International Conference on Mining Software Repositories, Austin, TX, USA, 14–15 May 2016; pp. 468–471. doi:10.1145/2901739.2903508. [CrossRef]
51. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine Learning in Python. *J. Mach. Learning Res.* **2011**, *12*, 2825–2830.