# Edge Computing Based Machine Learning Mobile Malware Detection

William G. Hatcher, Jarrett Booz, Josh McGiff, Chao Lu, and Wei Yu
Department of Computer and Information Sciences
Towson University, Towson, Maryland, USA 21252
{whatch2, jbooz1, jmcgif1}@students.towson.edu,{clu,wyu}@towson.edu

## ABSTRACT

Because network infrastructure and bandwidth cannot keep up with the increasing demands, edge computing has sprung up to reorganize the cloud computing paradigm. As an application of edge computing, mobile malware detection requires significant resources that are unavailable to individual devices. The immediacy of malware detection is highly pertinent, as malicious software may already be in operation. In addition, the dispersion of mobile devices and massive simultaneous traffic make the offloading this work to remote data centers infeasible. In our study, we implement a typical edge node to perform Android malware detection. In our study, we collect an exceptional amount of mobile malware and benign samples, and demonstrate the ability of edge computing nodes to provide typical security functions for edge devices. We also show the ability to scale the operation for increasing data volumes.

## Keywords

Edge Computing, Malware Detection, Machine Learning Techniques

## 1. INTRODUCTION

Edge computing, the paradigm to offload part or all of cloud computing to localized edge devices as needed, can provide improved performance when network infrastructure becomes a bottleneck to timely service. Similar to threat monitors or sensors widely deployed in Internet [30], edge nodes that deployed in various locations in the network, though not on par with datacenter hardware and processing power, can still easily serve large parallel applications and provide responsive service for complex computation. As an example, ability to massively scan and detect mobile malware is a pressing issue, as mobile smartphone, tablet, and Internet-of-Things (IoT) device adoption continues to increase. Particularly, malware detection on mobile devices faces unique challenges due to the resource-limited nature of mobile hardware. Additionally, the ability to handle immense volumes of applications simultaneously to serve the massive user base provides a unique challenge for cloud infrastructures.

The enormous growth in smartphone technology and widespread adoption has afforded significant benefits to economies worldwide. For example, the annual revenue of the worldwide mobile market in 2016 was approximately $ 1.05 Trillion [14]. Holding the largest market share of device operating systems, Google's Android OS reached 88 % in the third quarter of 2016, and has not dropped below 75 % of total mobile sales since late 2013, compared to Apple's iOS, which has failed to break 25 % in the same time frame [2]. Yet, as Android market share reaches record highs, malware dissemination and mobile vulnerabilities have likewise flourished, due in part to the open nature of the Android platform, the always-connected nature of mobile devices, and the massive adoption over other platforms. As the number of apps in the Google Play Store has reached more than 2.6 million, as of December 2016 [24], the opportunities for repackaging applications with malware, or circumventing app store security measures, become more and more viable. Thus, the accurate detection of mobile malware is thus a pressing issue.

The contributions of our work are summarized as follows.

- We propose an edge computing framework to provide latency-sensitive service to network edge devices. We then develop our proposed system architecture for an edge computing node, including the underlying services and the proposed testbed. The designed system, runs on Apache Hadoop, YARN, and Apache Spark to provide parallel computing and big data processing in a cloud environment. The outlined testbed is developed as a typical small-edge node, and our performance evaluation demonstrates the ability to scale the system.
- We develop and implement an Android malware detection system on our edge computing platform, utilizing Android application Permissions as detection features, and demonstrate its capacity to analyze big data. We collect a significant volume of Android malware and benign samples for machine learning-based detection. The samples are culled from multiple sources, and our evaluation encompasses one of the largest test datasets for Android analysis. We conduct a performance evaluation of the developed system, comparing the efficacy of the Naïve Bayes, Decision Tree, and Logistic Regression algorithms. Our results demonstrate similar results to other static analysis techniques that utilize permissions for malware analysis. In addition, our results demonstrate the scalability of our system in detecting malware across a massively large application dataset, having duplicated the data for big data testing.

The remainder of the paper is as follows: In Section 2, we provide background in Cloud and Edge computing, as well as mobile malware analysis. In Section 3, we describe our

proposed Edge Computing-based system architecture. In Section 4, we present our case study of an Edge Computing platform for conducting malware detection. In Section 5, we present the results of our performance evaluation. Finally, we conclude the paper in Section 6.

# 2. BACKGROUND

**Big Data, Cloud Computing, and Fog/Edge Computing:** In the interconnected world of Big Data and IoT, Streaming Analysis and Cloud and Fog/Edge Computing are positioned as key assets in support of these frameworks. Big Data, as the processing of data too large for traditional systems, requires massively distributed processing components to handle the data in a reasonable amount of time. For these reasons, the study of streaming data analysis, along with the other topics mentioned above, is widespread, and applies to every area of study.

Since the proposal of Fog computing by CISCO and Bonomi *et al.* [3] in 2012, research has flourished around the topic [17, 20, 5, 25], becoming the next great shift in the cloud computing ecosystem. In concept, the implementation of fog computing (also called edge computing) will realize the ability to direct long-term data dependent processing to the cloud, while simultaneously directing real-time low-latency processes to the network edge. Through massively deployed direct communications, Fog clusters localized to the network edge will handle low-latency processing, make cloud/fog computing decisions, perform load balancing, and provide inter-fog information transfer for continuation of mobile computing service.

**Mobile Malware Detection:** Extensive research has been carried out in the interest of detecting malware for the Android platform. Android malware detection can be carried out either statically, on the finite application package (APK), or dynamically on the target at runtime. Though static detection is computationally cheap, due to the target APK file being finite in size, static detection requires tools to decompile and extract meaningful features for analysis. These features can be Permissions to access system services, Intents that provide interprocess communication, API calls, traces of specific known malware families, etc. Static detection of Android malware has been studied from various perspectives [8, 6].

Dynamic detection instead extracts features from the application in operation, including power consumption, network traffic data, and system calls to the underlying Linux kernel. Dynamic detection, however, is significantly more complicated, as it requires data logging on a mobile device with root access, or a virtual machine running on non-native hardware, and must run for a suitable amount of time to capture appropriate behavior. A number of dynamic analysis have also been developed, often utilizing static analysis to support the detection [29, 26, 7]. This is often because the unpredictable nature of runtime events, while they may provide unique insight for detection, also can give rise to significant noise in the data.

# 3. EDGE COMPUTING-BASED SYSTEM

In this section, we outline the architecture, system elements, and our designed testbed for malware detection.

## 3.1 System Architecture and Elements

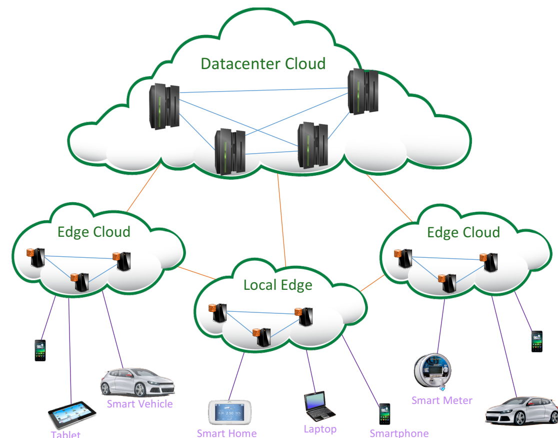


**Figure 1: Edge Computing Architecture**

Figure 1 illstrates the layout of the edge computing architecture, with edge cloud nodes supporting mobile devices, providing intercommunication between edges, and providing edge-to-datacenter communication for upstream offloading. In our study, we implement a typical Edge node, and perform what would be a typical edge node task: providing malware analysis and detection for mobile devices, as an example.

The primary software elements of the developed and implemented system are listed below.

***Apache Hadoop:*** The Hadoop Distributed File System (HDFS) uses several services to perform tasks on large amounts of data in a cluster environment. The driver service is NameNode. This service runs only on the master node and acts as the manager service for the entire file system, managing all of the data and all requests for client access to the file system. The other service used by HDFS is the DataNode service. The DataNode service runs locally on each of the nodes in the cluster. This service manages the blocks of data that are stored locally on each of the nodes.

In order for data to be processed close to where it is being stored in the cluster, HDFS breaks large files into smaller, equal sized blocks. These blocks are then distributed across the cluster. This is handled by the NameNode. The blocks are then stored locally on the cluster nodes. These blocks of data, as mentioned before, are managed locally on each of the nodes by the DataNode service. In theory, if the data is balanced across the cluster evenly, then most of the data will be as close as possible to the computation source, thus increasing the efficiency of the analysis [21].

***YARN:*** Another service that is part of the Hadoop Infrastructure is YARN (Yet Another Resource Negotiator), which serves as the resource management and job scheduling. The purpose for YARN is mainly to take the resource management and job scheduling components of distributed computing away from the processes that are actually processing the data. This separation allows Hadoop clusters to be more dynamic in the kinds of applications that they can run, as well as allows the clusters to run multiple jobs simultaneously.

YARN is comprised of two main features: the resource manager and the node manager. The main purpose of the resource manager is to control access of Hadoop system re-

sources. The resource manager runs on the master node for the cluster. The resource manager then sends out node manager agents to the worker nodes on the cluster. These node managers monitor and report on the processing operations occurring on each individual node.

When submitting an application to YARN, an application master is created on one of the worker nodes. This application master is responsible for arbitrating the resources allocated by the resource manager, as well as communicating with each of the worker nodes to determine application status. YARN must be configured and running on all nodes in the cluster in order to be able to submit applications to be run on the cluster via YARN. On each node, there is a configuration file, in which the administrator can override default settings that determine maximum and minimum resource allocations for the node. Adjusting the maximum and minimum resource allocation settings in this file can drastically impact the performance of the cluster [22].

***Apache Spark:*** Apache Spark is an engine for big-data processing. Spark has benefits in that it is fast in processing data. Spark also includes APIs with pre-built components for streaming services and machine learning for several languages, including Sala, Java, and Python. Spark performs tasks in processes called executors. The executor processes are housed within Spark Containers. Each executor will be assigned one or many tasks by the Spark Application Master and execute those tasks independent of the other executors. Results are then returned to and aggregated at the Application Master.

YARN submission is one way to run a Spark application. Other ways are Mesos and Spark Standalone submission. YARN submission has several advantages over the others. First, YARN submission is the only submission method that allows the administrator to choose how many executors to run on the cluster; other methods are required to run a single executor on every node in the cluster. YARN submission is also the only submission method that allows to secure, authenticated communication between executors, containers and nodes. Additionally, Spark submission via YARN allows for the full scope of resource management that YARN provides. This improves execution compared to the Spark Standalone or Mesos resource management services.

Spark runs across the Hadoop cluster when launching an application via YARN. First, a Spark application master will be spawned by YARN. Next, the application master can dynamically create YARN containers, which house Spark executors depending on the amount of resources that are required for completing the application. All Spark executors will operate locally and independent of the other containers and executors across the cluster [16, 4].

## 3.2 Testbed

The testbed environment that we design and use for analysis consists of a single master node and four slave (worker) nodes. Each node is configured with the following software: (i) *Ubuntu Server 16.04.2 LTS (x64):* Released April 21, 2016, Long Term Support Version of Ubuntu Server (Guaranteed support until April 2021), and Most recent stable release of Ubuntu Server LTS; (ii) *Apache Hadoop 2.7.3:* Released August 25, 2016 and most recent stable release for the Hadoop 2.7 line; (iii) *Apache Spark 2.1.0:* Released December 28, 2016, Most recent stable release, Pre-built Release for Hadoop 2.7; (iv) *Java SE Runtime Environment Version 8 Update 121:* Released January 17, 2017 and most recent stable release.

The Master node and the Slave nodes are run on a Dell PowerEdge T420 Server with virtualization software installed. The server CPU hardware consists of 2 Intel(R) Xeon(R) CPU E5-2407 v2 @ 2.40 GHz, equating to 8 physical cores with a total of 8 CPU threads. The Server had 94 GB of usable physical RAM. The specific virtualization software used is Proxmox Virtual Environment 4.4-1. A 64-bit Ubuntu Server 16.04 Virtual Machine (VM) is configured as previously described for the following nodes: Master, Slave-2, Slave-3, Slave-4. The master node is allocated 110 GB virtual hard drive storage, 16 GB RAM and 4 virtual processor cores. Each of the noted slave nodes is allocated 50 GB virtual hard drive storage, 16 GB RAM and 4 virtual processor cores. The remaining slave node (Slave-1) is hosted on a Dell OptiPlex 9010 Workstation. The workstation CPU is a single Intel(R) Core(TM) i5-3570 CPU @ 3.40GHz, providing 4 physical cores with 4 CPU threads.

# 4. CASE STUDY: EDGE COMPUTING-PLATFORM FOR MALWARE DETECTION

In the following, we present our case study, detailing the malware samples, detection methodology, and machine learning algorithms used for our edge computing cluster.

## 4.1 Malware and Benign Application Samples

In this case study, we amass a collection of Android applications, including both malware and benign samples for comparative analysis and machine learning assessment. Android malware, like any other type of malware, can be broken up into families based on the malware behavior and common features. Most malware families are some combination of ransomware, Trojan horse, botnet, adware, spyware, etc., and are implemented in various different ways. As mentioned above, the vast majority of Android malware are repackaged versions of popular applications, intended to trick unsuspecting users into downloading the malicious software from third-party app stores, or by providing difficult to distinguish clones. Malware samples collected between 2010 and 2011 produced some 49 distinct malware families, while the growth in Android malware throughout 2011 was exponential in nature [23]. As of 2017, there are now more than 175 known Android malware families [9], and ransomware is experiencing a new surge [10].

The entire dataset of malware and benign APKs tested comprises some 48,643 samples, from multiple sources. The primary sources of the malware samples are the now discontinued Android Malware Genome Project [15], the Contagio Mobile [13] malware repository, and the VirusShare [28] malware repository. Of particular note, the VirusShare repository contains over 28 million malware samples, covering all platforms, and adding more all the time. The number of unique malware samples used in our analysis is 17,280, collected between 2010 to 2016, and spanning a majority of Android malware families. Benign samples are downloaded from the Google Play Store, and the mirror websites APKMirror [12] and APKPure [1]. The benign sample set consists of 31,363 APKs. It is also worth noting that we confirm selected representative malicious and benign samples through submission to VirusTotal [27] virus scanning

service, a subsidiary of Google, that scans submitted files in more than 56 licensed antivirus software products.

## 4.2 Permission-based Detection

Permission-based detection has been widely demonstrated for Android malware detection. An Android .apk file (APK), the application installation files for Android operating system, is compressed files based on the JAVA .jar. The APK includes a file labeled AndroidManifest.xml, which defines the Permissions, among other things, for the Android application to access operating system services. By evaluating and comparing the requested permissions between malicious and benign applications, we can utilize machine learning tools to predict malicious applications.

For our implementation, Permission data is extracted from each APK file using the Android Asset Packaging Tool, AAPT. AAPT is a tool that can be used to unpack and unzip APK files in order to view the data inside the package. AAPT offers support for automatic permission data extraction upon successful unpacking. Output from AAPT for a particular app is logged to a text file for evaluation. The text is then transformed and formatted so that all data for a particular application is contained on a single line. This formatting is utilized for ease of analysis in Spark programs. In our experiment, we provide Permission data with malicious or benign application categorization as input to the chosen machine learning algorithms (outlined below) for training and testing. Some 19,197 unique permissions were identified in the sample set. These Permissions include both generic Android permissions as well as vendor/app custom permissions.

## 4.3 Machine Learning Schemes

We use several representative machine learning schemes to conduct malware detection.

***Naive Bayes:*** Generally speaking, the Naive Bayes learning algorithms are based on Bayes Theorem, and are classified as generative classification algorithms. These algorithms compute the probability that a piece of data fits a classification given a specific attribute value. All attributes of the data are considered equally when making a prediction because of the assumption that the value of a single attribute does not determine or affect the values of other attributes. A form of Naive Bayes called Multinomial Naive Bayes was used for our evaluation. This form of Naive Bayes is specifically tailored to text file analysis, and is included in the Apache Spark API. In this scheme, a priori probability calculation is performed to make a prediction on the data's initial classification. Next, the scheme reads in the remaining input strings as evidence for final classification. Lastly, a posteriori conditional probability, based on the gathered evidence and the priori prediction, is computed as used as the final prediction output [11].

***Decision Tree:*** Decision Tree is a classification algorithm that utilizes a tree data structure. The tree is comprised of smaller and smaller subsets of the larger dataset that can be used to make a decision. The tree consists of decision nodes and leaf nodes. Different attribute values make up the decision nodes. Attributes move from the general at the top of the tree to the more specific near the bottom. Based on the values of the attributes, the algorithm traverses the tree and finally lands on a leaf node. The leaf node contains the final classification for the piece of data. Each path from the root node to a leaf is known as a rule. Entropy calculations based on one and two attributes are used to quantify the information gain of each attribute in the tree. Entropy values of 0 indicate that a final decision can be made and therefore a leaf node is created. Entropy values greater than 0 require more splits in the tree structure [18].

***Logistic Regression:*** Logistic Regression is a prediction model for dichotomies, problems with exactly two outcomes. The logistic regression model is a curve constructed by computing the natural log of the odds (not probability) of a target variable(s). Any number of predictors can be used when computing the natural log function that is used to represent the data and these predictors do not have to have equal variance in the dataset. Related to this experiment, logistic regression has been used due to the fact that this classification (malware or benign) is a dichotomy. The Spark engine, when computing the model based on the training data, decides how many predictors to use and also computes the best fit logistic curve for the dataset [19].

# 5. PERFORMANCE EVALUATION

In the following, we describe the performance evaluation of our implemented edge computing node in malware detection of Android applications.

**Methodology:** The application that we have developed to perform the analysis on is written in Scala 2.10, utilizing the Apache Spark API. The application is designed to read in data from two text files, which contain Android application permissions data, one application's data per line of the text file. One file is dedicated to benign samples, and the other file is dedicated to known malware samples. This data is then converted to Scala Resilient Distributed Datasets (RDD) in order to be processed across the cluster. The RDDs are used as input to several aforementioned machine learning algorithms. All these machine learning algorithms are predefined in the Apache Spark API.

To prepare input data for processing Spark's built in Term Frequency Hashing function is utilized. Each line of text from the input files contains the permissions information for exactly one Android app. Using the Spark's hashing algorithm, each line is quickly converted into a feature vector, which is subsequently assigned a label value: 1 for a vector representing a benign sample, and 0 for representing a malicious sample. The RDD's containing these Labelled Point vectors are subsequently combined into our full dataset, and randomly divided up into two sets of vectors.

Each algorithm has been performed five times on the entire dataset utilizing four different training to testing ratios. Training to testing ratios used were: 20 % : 80 %, 40 % : 60 %, 60 % : 40 %, and 80 % : 20 %. These tests add up to a total of 20 iterations per algorithm. After each iteration, output is produced to record the following data points: accuracy in detection, rate of false positive, and rate of false negative.

Overall accuracy is computed by comparing the known value (malware or benign) to the value predicted by the algorithm. The number of samples that are correctly identified is then divided by the total number of samples that are tested in that particular iteration. The number of tested samples did not include the samples that are used to train the model; only the number of samples that the model is tested against are included for this count.

False positive and false negative rates are defined met-

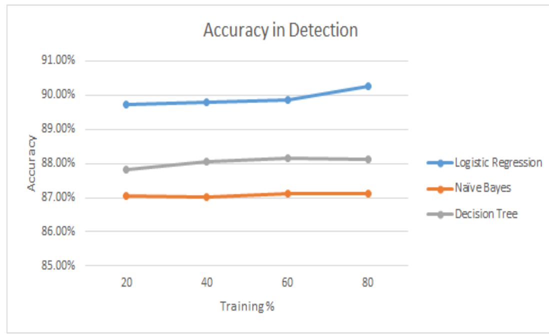


**Figure 2: Detection Accuracy vs. Training Ratio**

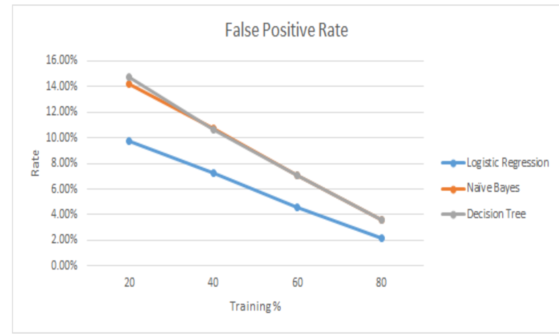


**Figure 3: False Positive Rate vs. Training Ratio**

rics for incorrectly classified samples. Generally speaking, false positive rate is computed by counting the number of known benign samples that are predicted by the algorithm to be malicious, then dividing by the number of samples that are known to be benign in the testing data. False negative rate is computed by counting the number of known malware samples that are predicted by the algorithm to be malicious, then dividing by the number of samples that are known to be malicious in the testing data. The number of tested samples do not include the samples that are used to train the model; only the number of samples that the model is tested against are included for this count.

In addition to the aforementioned test on the original sample set, another test is run on much larger sets of duplicate data. The original data set is duplicated 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024 times to produce much larger data sets. These larger sets are used to test run time of each of the algorithms. The original set of data is determined to be insufficient to test run time across the cluster, and therefore, the duplicate sets are made. The run time for each of the larger datasets is computed by timing how long it took for each iteration of the algorithm at 80 % training ratio.

**Evaluation Results:** We now present the results of our performance evaluation. Our results show logistic regression to be the most successful predictor of malicious content. Logistic regression peaked at a 90.26 % accuracy level for correctly predicting the classification of samples when trained on 80 % of the dataset.

Across the four different training/testing ratios, there is an almost negligible gain in accuracy. This is contrary to the expected results wherein it is predicted that accuracy would steadily increase as the models began to train on a larger portion of the data set. This may be indicative of the inherent limitations of using permissions alone to predict the classification of an app. It is possible that using a relatively small number of features to classify such a comparatively large dataset is insufficient to achieve an accuracy rate that is any higher.

It is clearly evident that the training to testing ratio had a distinct impact on both false positive and false negative rates. With a higher training ratio, it is expected that a lower number of false positives and negatives would result. This is precisely what the results indicate. In relation to false categorizations, there is about a 1 % lower rate of false negative than false positive.

Logistic Regression is the best performer with respect to detection rate, false positives, and also performs well in the

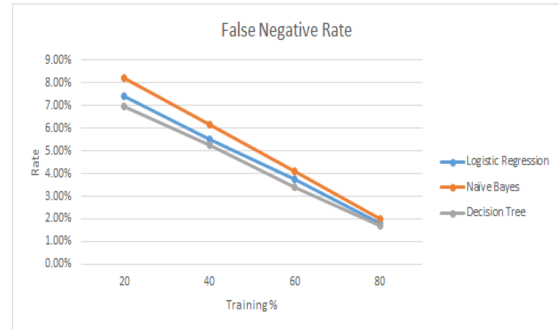


**Figure 4: False Negative Rate vs. Training Ratio**

false negatives test. This is expected because of the nature of the algorithm. As described earlier, logistic regression is specifically designed to handle binary data classification.

To measure the scalability of the designed system, each algorithm is tested against increasingly large datasets comprised of duplicated data from our initial set. As seen in the figure, performance scales very favorably at the smaller sample sizes. Given the negligible increase in run time at these smaller datasets, we assume that the majority of the run time on these samples is spent on cluster overhead. Only once we reach our data set scaled by a factor of 128, we can see the first noticeable climb in run time. From this point, there is a roughly linear increase in run time, corresponding to the increase in increasing size of our dataset.

# 6. FINAL REMARKS

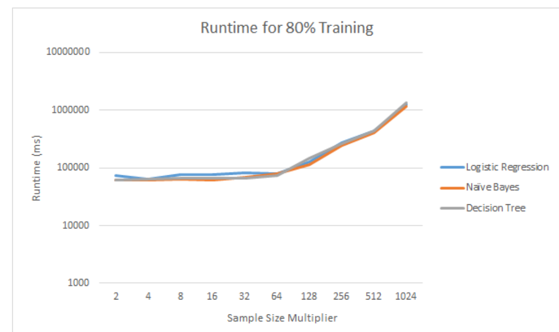


**Figure 5: Running Time vs. Sample Sizes**

In this paper, we addressed the issue of detecting malware using edge computing-based infrastructure. Particularly, we implemented a typical edge node to perform Android malware detection. We collected an exceptional amount of mobile malware and benign samples, and demonstrated the ability of edge computing nodes to provide typical security functions for edge devices. We also showed the ability to scale the operation for increasing data volumes. Based on our experimental results, it can be concluded that utilizing edge computing to analyze permissions data in an attempt to determine whether an Android APK could be considered malicious is reasonably effective. Run time in the cluster environment proved to be highly efficient and scalable until the maximum cluster efficiency threshold was reached.

## Acknowledgment

The project sponsored by the National Security Agency is under Grant Number H98230-15-1-0272. The United States Government is authorized to produce and distribute reprints notwithstanding any copyright notation here.

## 7. REFERENCES


[1] apkpure.com. https://apkpure.com/.

[2] A. Bhattacharya. *Android just hit a record 88 % market share of all smartphones.* https://qz.com/826672/android-goog-just-hit-a-record-88-market-share-of-all-smartphones/, November 2016.

[3] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proc. of the ACM First Edition of the MCC workshop on Mobile cloud computing*, 2012.

[4] I. Cloudera. *Running Spark Applications on YARN.* https: //www.cloudera.com/documentation/enterprise/5-8-x/topics/cdh_ig_running_spark_on_yarn.html.

[5] R. Deng, R. Lu, C. Lai, T. H. Luan, and H. Liang. Optimal workload allocation in fog-cloud computing towards balanced delay and power consumption. *IEEE Internet of Things Journal*, 3(6):1171–1181.

[6] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, and S. Furnell. Androdialysis: Analysis of android intent effectiveness in malware detection. *Computers & Security*, 65:121–134, 2017.

[7] D. Geneiatakis, I. N. Fovino, I. Kounelis, and P. Stirparo. A permission verification approach for android mobile applications. *Computers & Security*, 49:192–205, 2015.

[8] I. Gurulian, K. Markantonakis, L. Cavallaro, and K. Mayes. You can?t touch this: Consumer-centric android application repackaging detection. *Future Generation Computer Systems*, 65:1–9, 2016.

[9] X. Jiang and Y. Zhou. *Android Malware.* Springer-Verlag New York, 2013.

[10] M. LABS. *State of Malware Report.* https://www.malwarebytes.com/pdf/white-papers/stateofmalware.pdf, 2017.

[11] C. D. Manning, P. Raghavan, and H. Schütze. *Naive Bayes text classification.* https://www.virustotal.com/, 2009.

[12] A. Mirror. https://www.apkmirror.com/.

[13] C. Mobile. https://contagiominidump.blogspot.com/.

[14] K. Okeleke, M. Rogers, and X. Pedros. *The Mobile Economy 2017.* https://www.gsmaintelligence.com/research/2017/02/the-mobile-economy-2017/612/, February 2017.

[15] A. M. G. Project. http://www.malgenomeproject.org/.

[16] S. Ryza. *Apache Spark Resource Management and YARN App Models.* https://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/, May 2014.

[17] S. Sarkar, S. Chatterjee, and S. Misra. Assessment of the suitability of fog computing in the context of internet of things. *IEEE Transactions on Cloud Computing*, 2015.

[18] S. Sayad. *Decision Tree.* http://www.saedsayad.com/decision_tree.htm, 2017.

[19] S. Sayad. *Logistic Regression.* http://www.saedsayad.com/logistic_regression.htm, 2017.

[20] M. Slabicki and K. Grochla. Performance evaluation of coap, snmp and netconf protocols in fog computing architecture. In *Proc. of IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2016.

[21] A. Software. *Apache Hadoop 2.7.3.* http://hadoop.apache.org/docs/r2.7.3/.

[22] A. Software. *Apache Hadoop YARN.* http://hadoop.apache.org/docs/r2.7.3/hadoop-yarn/hadoop-yarn-site/YARN.html.

[23] I.-C. Spreitzenbarth. *Current Android Malware.* https://forensics.spreitzenbarth.de/android-malware/, January 2016.

[24] Statista. *Number of available applications in the Google Play Store from December 2009 to December 2016.* https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, 2017.

[25] I. Stojmenovic and S. Wen. The fog computing paradigm: Scenarios and security issues. In *Proc. of IEEE Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2014.

[26] F. Tong and Z. Yan. A hybrid approach of mobile malware detection in android. *Journal of Parallel and Distributed Computing*, 103:22–31, 2016.

[27] V. F. O. Virus. *Malware and URL Scanner.* https://www.virustotal.com/.

[28] VirusShare.com. https://virusshare.com/.

[29] X. Wang, W. Yu, A. Champion, X. Fu, and D. Xuan. Detecting worms via mining dynamic program execution. In *Proc. of International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2007.

[30] W. Yu, X. Wang, X. Fu, and W. Z. D. Xuan. An invisible localization attack to internet threat monitors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 20(11):1611–1625, 2009.