Health State of Google's PlayStore
Finding Malware in Large Sets of Applications from the Android Market

Alexandre Dey, Loic Beheshti and Marie-Kerguelen Sido
Operational Cryptology and Virology Research Laboratory $(C+V)^o$, ESIEA,
38 rue des Docteurs Calmette et Guérin, Laval, France

Keywords: Android, Malware, Machine Learning, Crawler, Neural Network, Static Analysis.

Abstract: Android has, to this day, more than 80% of the mobile OS market share. Android users also have access to more than 2 million applications via the Google Playstore. The Playstore being an official market, users tend to trust the applications they find in it, and therefore, the store is an interesting platform to spread malware. We want to provide a health state of this store by finding the proportion of malware that managed to get published in it. In this paper, we explain how we developed the crawler that massively downloads the application directly from the Playstore. Then we describe what features we extract from the applications and how we classified them with the help of an Artfificial Neural Network. Our study confirms that there are malicious applications on the Playstore. The proportion of them is around 2%, which corresponds to about 40,000 officially downloadable malware.

1 INTRODUCTION

Google, with its Android platform, has been solidly integrated into the smartphone market in recent years. One of the reasons for its success is its application catalog also called "Playstore" (Statista (2017)) which contains, based on our estimation, more than 2 million applications. It is also the developers' favorite platform because publishing applications is both quick and free.

However, the number of users of this platform make it an attractive target for spreading malware (malicious applications). Despite the protections put in place by Google, it is certain that malicious of applications exist on the Playstore such as Viking Horde (A. Polkovnichenko (2016)), DressCode (TrendMicroInc (2016)) or FalseGuide (O. Koriat (2017)) attacks have demonstrated.

This article will present our study which aims at determining the proportion of malware on Google Play. We consider as malware any application intentionally causing harm or subverting the system's intended function (this includes Adware), as well as manipulating information without user's express consent. To perform this study, we have designed a crawler able to simulate the download of an application and to bypass the downloading restrictions of Google Play. Then, with data from static analysis, we trained an Artificial Neural Network to discover new malware on the PlayStore. This neural network bases its detection on features that are extracted by reversing these applications.

In the following paper, we will detail the different techniques involved in performing our study. It is organized as follow : Section II, the crawler ; Section III, the features used for static analysis ; Section IV, the feed and forward neural network ; Section V the results. We will conclude by discussing our results.

2 PLAYSTORE CRAWLER

As we are conducting a study on Googles Playstore, we first need to retrieve a huge set of applications from it (large enough to be statistically representative). To do so, we have developed a Python script to download applications massively directly from the Playstore. We describe here the restrictions implemented by Google and how we bypassed them.

First of all, there is no exhaustive list of the applications available on the Playstore. Moreover, Google tends to bring out the most popular applications and to hide those less successful. Secondly, even though the communication with Playstore servers go through HTTPS, to only allow Android devices with a valid Google account, a proprietary protocol is used on top

Dey A., Beheshti L. and Sido M.
Health State of GoogleâĂŹs PlayStore - Finding Malware in Large Sets of Applications from the Android Market.
DOI: 10.5220/0006727705380544
In *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP 2018)*, pages 538-544
ISBN: 978-989-758-282-0

of it. This protocol is based on Google's protobuf (Google (2017)), and has not been officially described as of now, but has been, for the most part, reverse engineered (Figure 1). Finally, any behaviour considered as not "human" (such as downloading a large amount of applications in a short time, or following the same pattern at fixed interval, ...) will be detected and will lead to an account ban. Furthermore, the criteria to define whether or not a behaviour is normal seems to be changing every few weeks or months.

Previous work that involves crawling the Playstore has been made by other researchers, and they described how they prevent account banning. *Playdrone* (N. Viennot and Nieh (2014)) crawls the entire Store using Amazon EC2 cloud services and thus connections come from multiple IP addresses. As for *SherlockDroid* (Apvrille and Apvrille (2014)), they limit the number of downloads by pre-filtering what they are interested in (potential malware), and also, each connection goes through Tor. The first one is expensive and the second one does not respond to our problematic, crawling the whole PlayStore.

```
message AndroidAppDeliveryData {
  optional int64 downloadSize = 1;
  optional string signature = 2;
  optional string downloadUrl = 3;
  repeated AppFileMetadata additionalFile = 4;
  repeated HttpCookie downloadAuthCookie = 5;
  optional bool forwardLocked = 6;
  optional int64 refundTimeout = 7;
  optional bool serverInitiated = 8;
  optional int64 postInstallRefundWindowMillis = 9;
  optional bool immediateStartNeeded = 10;
  optional AndroidAppPatchData patchData = 11;
  optional EncryptionParams encryptionParams = 12;
}
```

Figure 1: Playstore's protocol part (application download).

On GitHub, we can find a Python script (dflower (2014)) that is able to communicate with the Playstore. This script implements the basic functionalities of the store (search for applications, get the details, download, ...) and needs only two elements to work with: a valid Google account and an Android Device ID (a unique token generated for each Android device) (Android (2017c)). However, the tool is not designed for mass downloading and we modified it to suit our needs.

To avoid being banned because of non-human behaviours, we crowd-sourced the creation of 30 activated Google accounts (an Android device is required to validate an account before it is allowed to download from the Playstore, and you can only create a few accounts from one device) and we try to put as much time as possible between two downloads from the same account (multiple minutes). We also generate an Android ID for every account thanks to *android-checkin* (Viennot (2012)). When crawling, we keep the authentication token as long as it is possible to download applications and we renew the token each time it is invalidated. When an account is banned, the Playstore servers return an error message when attempting to login. We use a list of HTTPS proxies to prevent our IP address from being blacklisted.

We find the applications by using the search function of the Playstore with words randomly selected from multilingual dictionary. Each search gives at most 250 results. From this point, we download applications that are free (we cannot download paid apps) and "compatible" with our fake device (we cannot download tablet specific apps without a compatible Android Device ID). To speed up the whole process, we split the dictionary between multiple processes and each of these spawns a thread provided with an account/device ID pair for each app to download (see Figure 2).

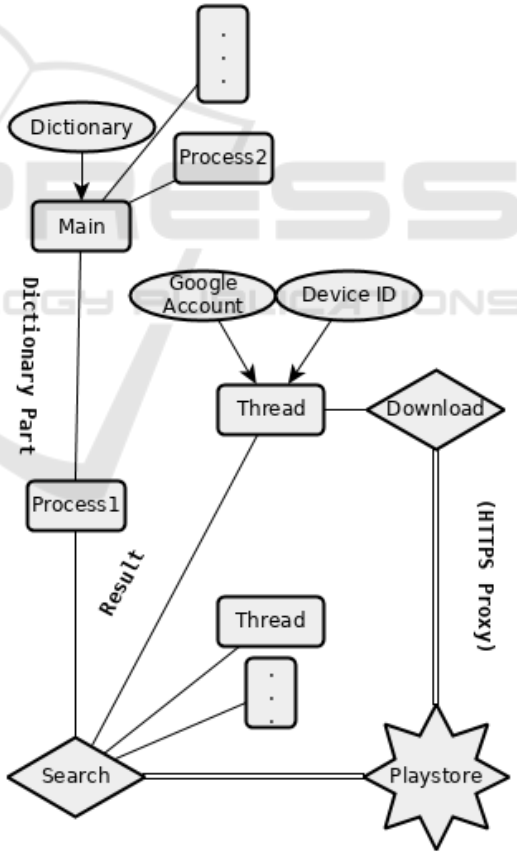

Figure 2: Playstore Crawler scheme.

The crawler is designed to be scalable. Provided that we have enough accounts and Device IDs, it is possible to distribute the crawling across multiple machines. In our case, to synchronize the different pro-

cesses, we store the applications inside a Cassandra NoSQL database (Apache (2008)) and before downloading a new one, we check if the application is absent from the database.

The crawler has been used for two purposes. First, in October 2016, it was configured not to download applications, but just to find them. It has generated a list of more than 2 millions applications. In January 2017, we used it to download and extract the features (see section 3) of 3,650 applications, which constitute our sample for this study. For this second usage, we used a laptop with a 2 core 4 thread intel CPU and the process took 2 weeks (depending on the application extracting the features can take up to 15 min while downloading takes less than one minute).

3 FEATURE ANALYSIS

3.1 Extraction

We perform static analysis, and therefore, we extract all the features that can be interesting for classification directly from the APK file. To reverse applications, we use the set of tools Androguard (Bachmann (2015)) written in Python. We provide the type of features we extract and why. Our static analysis follows a similar pattern to what was presented by P. Irolla and E. Filiol at Black Hat Asia Conference (2015) (Irolla and Filiol (2015)).

- **General Information:**

  This information is not directly used to perform the analysis, but more as a way to identify and contextualize it. We extract these information mainly from the application manifest (Android (2017a)).

  First, we retrieve the application common name (ex: Facebook Messenger) the package name (com.orca.facebook), and the version number. Both of these are supposed to identify the application, but considering it is easy to repackage an Android application, we use the SHA-256 hash of the APK instead.

  We also extract the certificate used to sign the application (to be published on the Playstore, an application must be signed). From this certificate, we can extract interesting information on the developer.

  The other things we extract are the information relative to the SDK (Android (2012)) (minimal/-maximal/target version), the intent for the statically defined receivers (communication between applications), intents for the activities (starting a foreground process inside an application) and intent for the services (background processes). The last thing retrieved is all the URLs statically defined in the application.

- **Classification Information:**

  These are the features on which the detection is based. They are retrieved by decompiling the .dex files in the application. These files store the java bytecode that will be executed by dalvik (or Android RunTime), the android Java virtual machine (Android (2017b)). We chose these features because, based on other researcher's work, they seemed promising for static analysis (see: G. Canfora and Visaggio (2015b); G. Canfora and Visaggio (2015a); D. Arp and Rieck (2014)).

  Most of the information we extract is based on the opcodes (the instructions executed by dalvik) themselves, without the operands. The first thing we use is opcodes frequencies (G. Canfora and Visaggio (2015b)). As a more representative value, we store trigrams (An N-gram is a sequence of N adjacent opcodes) frequencies (G. Canfora and Visaggio (2015a)). In Figure 3, the opcodes are (*invoke-static, move-result-object, if-eqz, invoke-direct, return-object, const/4, goto*), and the corresponding trigrams would be (*[invoke-static:move-result-object:if-eqz], [move-result-object:if-eqz:invoke-direct], [if-eqz:invoke-direct:return-object], [invoke-direct:return-object:const/4], [return-object:const/4:goto]*)

  To use several Android API methods, Android applications have to ask for certain permissions (use network, bluetooth, access user information, ...). Malware also have to ask for these permissions, and some permissions give access to more potentially malicious behaviour, and are therefore a good way to analyze applications (D. Arp and Rieck (2014)).

  Finally, we extract the Android API call sequence. The API is a software interface (a set of functions) used by developers to perform device related tasks (sending SMS, manipulating the UI, etc...). Some of these tasks can be maliciously employed. These calls were found to also be pertinent to detect android malware (D. Arp and Rieck (2014)). For Figure 3, API call would be *api_function_call1*, *api_function_call2*, as well as the constructor of *api_type1*

From a collection of applications, we extracted these features. There is a total of 228 dalvik opcodes. The applications were constituted on average of 80,000 to 100,000 opcodes, but a few reached more

```
method_name
{
  invoke-static {v3}, api_function_call1
  move-result-object v0
  if-eqz v0, 000c
  new-instance v1, api_type1
  invoke-direct {v1, v0}, api_function_call2
  return-object v1
  const/4 v1, #int 0
  goto 000b
}
```

Figure 3: Decompiled code (SMALI).

than a million of them. From these opcodes, we found a list of 500,000 trigrams, but not all trigrams are present in each application. For the API calls, we recorded a total of 100,000 different ones. Finally we found 250 permissions. Therefore, if we wanted to represent applications inside vectors, the dimension of them would be greater than 600,000.

3.2 Feature Selection

Even with neural networks being powerful tools, we must limit the number of entries to have optimal results. Taking as entry all the application's features would lead building a neural network with millions of entries which is not easily computable. An optimal way to limit our entries is to take the most relevant information/features, those features can be found using statistical methods.

A way to find relevant features would be to find the features most represented in Malware. A classic way to do this is to first apply Principal Component Analysis (PCA). PCA was not implemented for our problem for many reasons. The main reason is the amount of data, PCA has a too high complexity, $O(f^2n + f^3)$, where f is the number of features and n the number of samples, and this is too difficult to solve considering the number of features we have.

To optimize results on malware detection, we took the features most represented in malware while being the least represented in benign applications. A well-known statistical method allows us to find those features easily, the *TF-IDF* indicator (J.Z. and Maloof (2006)). This method, originally used for document research for example in search engines, can be adapted for our purpose. The *TF* or term frequency is the frequency of a word in a given text. The *IDF* or inverted document frequency is a value evaluating the generality of a given sequence, the more a word is represented, the less pertinent it becomes as a matter of selection, the lower the *IDF* value is.

The *TF-IDF* is the product of these two values, thus, the higher the value is the most pertinent the result will be. In our case, things are slightly different. *TF* is the feature presence frequency in our malware database and the *IDF* is the feature presence frequency in the benign dataset. We have then an indicator that grows if the feature is well represented inside the malware database and diminish if the feature is well represented in the benign application database.

$$tf(t,d) = \frac{f_{t,d}}{\sum_{t' \in d}(f_{t',d})} \quad (1)$$

$$idf(t,d) = \frac{N}{|\{d \in D : t \in d\}|} \quad (2)$$

$$tfidf(t,D) = tf(t,d) * idf(t,D) \quad (3)$$

After calculating our modified version of the *TF-IDF* on each feature and sorting the results from the most pertinent to the least pertinent, we plot the *TF-IDF* histogram for each section (Figures 4, 5 and 6). We can select the most important features given on the results distribution and, by cutting a section for each graph, we then obtain 12175 features which will be our entries.

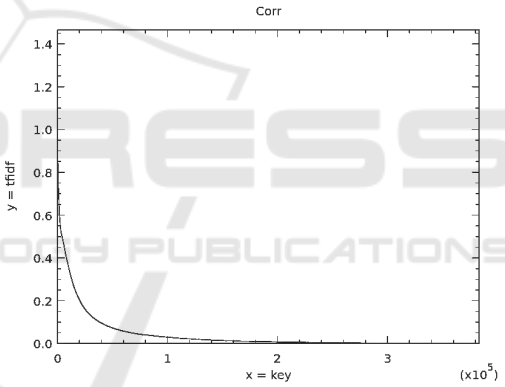

Figure 4: *TF-IDF* curve: Trigram Frequencies.

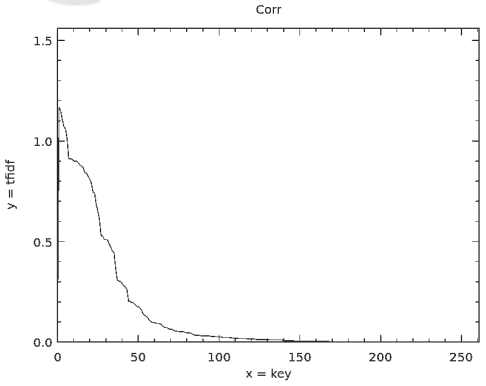

Figure 5: *TF-IDF* curve: Applications Permissions.

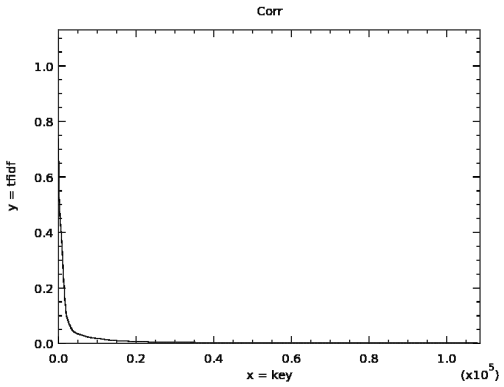

Figure 6: *TF-IDF* curve: API Calls.

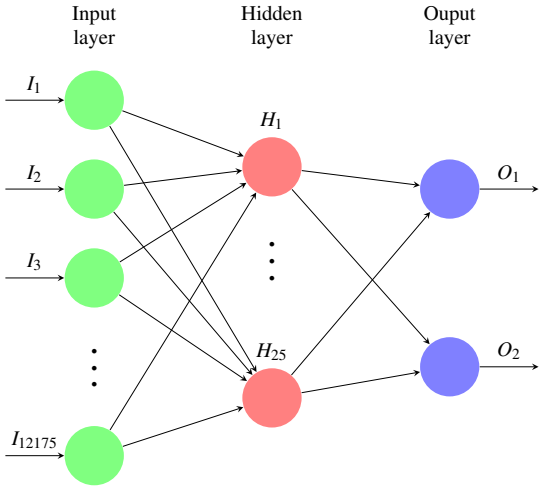

Figure 7: Multilayer Perceptron.

4 CLASSIFICATION

4.1 Training Dataset

We use an artificial neural network to classify applications. To build it, we need a set of applications that are already classified. We use the Drebin Dataset (D. Arp and Rieck (2014)). The Drebin Dataset is widely used by the Android malware researcher community.

At first we used the 5585 malware and 2201 benign applications as our training dataset. We have conducted a study on this initial set (Irolla and Dey (2017)) that show that 42.5% of these applications can be regrouped into 592 (584 malware, 8 benign) sets of repackaged applications with the same code and only resources and a few strings that differ from one another. Therefore, we removed the duplicates from the Drebin Dataset, and we end up training on a set constituted of 2891 malware APK and 2178 benign ones.

4.2 Artificial Neural Network

As for the classifier, we use an Artificial Neural Network (ANN) (Widrow and Lehr (1990)). Our neural network is a multilayer perceptron. A perceptron is one of the simplest structures in machine learning but, from previous experimentation using a single perceptron applied to malware detection, we deduced that this structure was sufficient for our problematic.

The final neural network parameters are: 1 input and 1 hidden layer set to a sigmoid symmetric activation function , 12175 entry nodes, 25 nodes on the hidden layer, 2 nodes on the output (Figure 7), training results are (-0.95;0.95) for a malware and (0.95;-0.95) for a benign application. The learning method is incremental (classic backpropagation), the learning rate is set to 0.25 and the library used is FANN (Nissen (2015)). The entry nodes will take the selected features frequencies as entries. The hidden layers are the layers in between the entry layer and the output, the more we have nodes and layers on the hidden layer, the more complex the structure becomes to train. The results are the values on the two output nodes, our decision strategy is *winner takes all*.

The neural network configuration was a compromise between performance and training speed.The learning rate can be seen as the progression speed, but a too big learning rate may miss the optimal solution. The neural network is trained to maximize the generalization property via cross validation. The validation subset is 20% of our original subset of 4908 applications.

4.3 Testing Phase

We trained our neural network to maximize our generalization capabilities. The optimal training was found thanks to the cross validation method. Our criteria for this training is the MSE (Mean Squared Error) (Lehmann and Casella (1998)). We select the training stage where the MSE on a foreign dataset is at its lowest (See Figure 8).

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\widehat{e_i} - e_i)^2 \tag{4}$$

The best neural network among all the created neural networks is the one with the best accuracy. The accuracy criteria is:

$$Acc = \frac{TP+TN}{TP+TN+FP+FN} \tag{5}$$

With TP, number of True Positive, TN, True Negative, FP, False Positive and FN, False Negative. For

a health state, instead of a ROC analysis, we privileged accuracy with an ANN giving a 98.03% accuracy with equal false positive and false negative rates. As such, with a large enough sample, applications falsely detected as malware are compensated by the malware that are not detected, and thus, the proportion of malware is accurate.

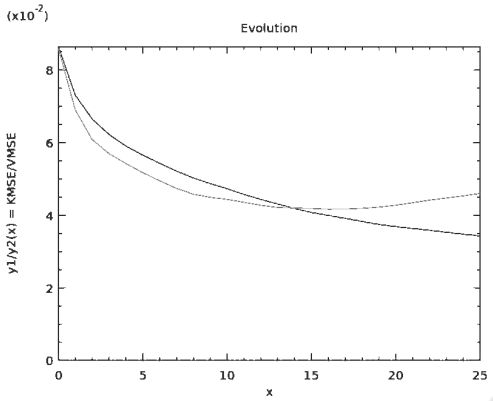

Figure 8: MSE for Generalization VMSE (Gray) and Knowledge KMSE (Black).

5 RESULTS AND CONCLUSION

We ran the crawler for 2 weeks on a machine shipped with an Intel®Core^{TM} i7 3520M CPU. Each application was decompiled before downloading the next one, and that was what took the majority of the time. We harvested a total of 3,650 applications directly from the Playstore. We search the Playstore for applications using random words in a large dictionary and therefore these 3560 applications constitute a representative sample. Before downloading, we also used the crawler to generate a list of applications on the Playstore. This list contains a little under 2 million applications.

Among the 3,650 applications, the neural network classified 92 as being malicious. This represents 2.52% of our sample. From this, and considering that our classifier is selected to be the most accurate and that false positive rate is roughly equal to false negative rate, assuming proportion of malware follows a normal distribution, using the normal estimation of the interval (6), we calculate the confidence interval of the proportion of malware on the Playstore:

$$[X - 1.96\sqrt{\frac{X(1-X)}{n}}, X + 1.96\sqrt{\frac{X(1-X)}{n}}] \quad (6)$$

With X our measured proportion and n the sample size. We can estimate with a 95% confidence level that the proportion of malware on the Playstore sits between 2.01% and 3.03%.

Some of these applications were manually tested and proved to be malicious. For example, the application *Battery King* (now removed from the store) is granted many permissions by the user and uses them to leak information about him. The application also writes binary file on the machine and execute them afterward.

By allocating more resources (more accounts, Device IDs, CPU power, ...) to the crawler, it would be feasible to crawl most of its applications. The main bottleneck here is the time needed to extract the features from the applications. This is explained by the fact that Androguard is a really complete tool that does more than what we actually need. Some testings showed that using a C++ program greatly reduces the extraction time for the opcodes (this program is still work in progress). Additionally, once it will be easily deployable and fully documented we plan on releasing the crawler to the OpenSource community (https://github.com/AlexandreDey/cygea-playstore-crawler).

Considering the classifier, it shows great accuracy for only light computation time. The *TF-IDF* variant employed here also allowed us to target more precisely the important features without the need of deep knowledge in the Android malware domain. However, moving to a deep learning algorithm might be beneficial for the accuracy rate. Classification capabilities could be enhanced by using a larger training dataset constituted of more actual and complex malware.

Finally, we are sure that Google's Playstore houses a certain number of malware, and we estimate that this number is between 40,000 and 60,000 applications. We are hoping that in the near future it will be possible to conduct the same kind of study on a larger scale, and with the best classifier possible.

ACKNOWLEDGMENT

We would like to thank Paul Irolla, currently pursuing a PhD at $(C+V)^o$ for his knowledge and support throughout the whole study. We also thank Eric Filiol, head of the $(C+V)^o$ laboratory for making this possible.

REFERENCES

A. Polkovnichenko, O. Koriat, V. H. (2016). A new type of android malware on google play.

Android, D. (2012). Developer tools. https://developer.android.com.

Android, D. (2017a). App manifest. https://developer.android.com/.

Android, D. (2017b). Dalvik executable format. https://source.android.com.

Android, D. (2017c). Setting.secure android id. https://developer.android.com.

Apache (2008). Apache cassandra. http://cassandra.apache.org/.

Apvrille, A. and Apvrille, L. (2014). Sherlockdroid, an inspector for android marketplaces.

Bachmann, S. (2015). Androguard. https://github.com/androguard/androguard.

D. Arp, M. Spreitzenbarth, M. H. H. G. and Rieck, K. (2014). Drebin: Effective and explainable detection of android malware in your pocket.

dflower (2014). google-play-crawler. https://github.com/dflower/google-play-crawler.

G. Canfora, A. De Lorenzo, E. M. F. M. and Visaggio, C. (2015a). Effectiveness of opcode ngrams for detection of multi family android malware.

G. Canfora, F. M. and Visaggio, C. (2015b). Mobile malware detection using op-code frequency histograms.

Google, D. (2017). Protocol buffers. https://developers.google.com/protocol-buffers/.

Irolla, P. and Dey, A. (2017). The duplication issue within the drebin dataset. Submitted to Journal of Computer Virology and Hacking Techniques.

Irolla, P. and Filiol, E. (2015). (in)security of mobile banking...and of other mobile apps.

J.Z. and Maloof, M. (2006). *Learning to Detect Malicious Executables*.

Lehmann, E. and Casella, G. (1998). *Theory of Point Estimation (2nd ed.)*.

N. Viennot, E. G. and Nieh, J. (2014). A measurement study of google play.

Nissen, S. (2015). Fast artificial neural network library. https:// github.com/libfann/fann.

O. Koriat, A. Polkovnichenko, B. M. (2017). Falseguide misleads users on googleplay.

Statista (2017). Number of available applications in the google play store from december 2009 to september 2017. https://www.statista.com.

TrendMicroInc (2016). Dresscode and its potential impact for enterprises. http://blog.trendmicro.com.

Viennot, N. (2012). Android checkin. https://github.com/nviennot/android-checkin.

Widrow, B. and Lehr, M. (1990). 30 years of adaptive neural networks: perceptron, madaline, and backpropagation.