App behavioral analysis using system calls

Prajit Kumar Das, Anupam Joshi and Tim Finin
Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County
{prajit1, joshi, finin}@umbc.edu

***Abstract*—System calls provide an interface to the services made available by an operating system. As a result, any functionality provided by a software application eventually reduces to a set of fixed system calls. Since system calls have been used in literature, to analyze program behavior we made an assumption that analyzing the patterns in calls made by a mobile application would provide us insight into its behavior. In this paper, we present our preliminary study conducted with 534 mobile applications and the system calls made by them. Due to a rising trend of mobile applications providing multiple functionalities, our study concluded, mapping system calls to functional behavior of a mobile application was not straightforward. We use Weka tool with manually annotated application behavior classes and system call features in our experiments, to show that using such features achieves mediocre F1-measure at best. Thus leading to the conclusion that system calls were not sufficient features for mobile application behavior classification.**

I. Introduction

Mobile devices have become ubiquitous in today's world due to their power, convenience and low cost. They have moved on from serving a singular goal of enabling voice communication to becoming the primary form of interaction with the information infrastructure. They act as an alternate form of identity for a user and are used as means of financial transactions (e.g. Google Wallet, Apple Pay, Banking apps). They also provide a convenient storage system for users' personal, private and confidential data and contain a collection of precise sensors making them a treasure trove for anyone who wishes to obtain users' private information or monitor activities, location and habits.

Although mobile devices are essentially personal, the rise in Bring-Your-Own-Device (BYOD) policies in the corporate domain, that permit user-owned devices to be used both within and outside a corporate firewall makes it easy to exfiltrate sensitive information. A study [1] from November 2014 found that 74% of organizations they researched allow their employees to bring their own devices in to company premises, and that 60% of all organizations allowed employees to use personal devices to access company networks and data. An additional 14% planned to allow the same within a year. The potential attack surface opened by such policies has been well recognized. These challenges lead to a critical need for behavioral monitoring of mobile applications (hereafter simply called as *apps*) to detect deviation from expected behavior that might indicate adverse intent.

Due to such threats and the traditional usage of system calls to monitor programs in computing systems [2] we came up with the following research question:
RQ1: Can system calls be used to distinguish between how an app "behaves" and its perceived/stated purpose?

Consider the "Brightest Flashlight App", which was sanctioned [3] in 2013 for collecting user locations and surreptitiously uploading the same to an advertising network. In this case, the behavior of a flashlight application on a mobile device should be focused on camera API access to turn the flash on or off. However, this app did more than such an expected behavior and collected location information of users. Determining whether an app's behavior was indeed legitimate requires in-depth analysis and understanding of app behavioral patterns and matching them to expected behavioral patterns. Therefore, we looked at system call analysis as a means of behavioral pattern recognition and the study conducted by [2] led to our assumption that system calls could help us extract features to be used in machine learning classifiers for app behavior classification. So, for instance, if we could classify an app as a flashlight app, such an app should only make system calls related to the camera API. The above app however, would make additional system calls related to GPS and Networking.

In this paper, we present the results of a preliminary study conducted with 534 Android apps from the Google Play Store [4]. System calls made by the apps were recorded and four different classifiers were used to classify apps into their behavioral classes. App behavior classes were determined using manual annotation carried out after using an app on an emulator. A secondary classification problem we studied, used Google Play Store [4] assigned categories for the apps. Through our study, we show that given the presence of complex app functionality, system calls were not sufficient to be used as the only features in classification of app behavior. We also show that system calls do not lead to good classification accuracy when using Google's class labels either.

The main contributions of this paper include a detailed analysis of system calls as features for behavioral classification of mobile apps. We attempt to match such behavior to expected or stated purpose of an app through manual annotations. We also present and discuss various issues arising from running such a study with real applications and real devices. We explain an alternate method of using emulators for carrying out such a study and constraints of an emulator based study.

Fine-grained access control on mobile devices in a BYOD usage context was researched in the past [5], [6]. Although access control solutions exists from research projects or open source projects [7] or even enterprise solutions from Google [8], access control policy generation remains an open problem. Determining an access control policy for an app could be driven by how it behaves in certain contextual situations. For example: it might be permissible to send some data over the corporate VPN, but not have the data uploaded to Facebook. In a BYOD scenario, a priori knowledge of Facebook's behavior pattern of automatically uploading pictures to its server could be critical to the process of policy generation. Our research could potentially be used to augment the process of access control policy generation.

Our literature survey on system calls revealed their traditional use in software behavior study [2]. In the mobile software (apps) domain, we found three categories of research work. First was malware classification [9], [10], second was use of NLP techniques for app behavior study [11], [12] and third was tracking of sensitive data on mobile device [13], [14]. To the best of our knowledge, none of the research work has focused on matching system calls to expected behavior of apps.

In Section II we discuss, the related works in further detail. We describe our system overview in Section III. We present technical details of our system and experimental setup in Section IV. Experimental results are presented in Section V. Finally we conclude our paper with a summary and examination of future directions.

II. Related Work

An important issue with privacy preservation on mobile devices stems from the fact that users tend to be privacy pragmatists [15]. Although any user would prefer that their data remain secure and private, the moment they realize the potential advantage of using an application, they choose to ignore such preferences. One way to ensure user data privacy and security would be to educate them about apps and the potential harm they may cause. However educating users would at first necessitate that we understand the potential harm an app may cause. Our study is an attempt to gather such knowledge.

The study by Kosoresow et. al. [2] led to our notion that system calls can potentially help detect app behavior patterns. Consequently, we have attempted to achieve the complicated goal, of app behavior classification, in our work. We consider this task to be a complicated when compared to malware classification due to the fact that, it is sometimes difficult to determine if an app's behavior was a legitimate functionality or an illegitimate behavior. We attempted to capture this distinction in our annotation process.

Research in mobile app analysis domain has shown emphasis on three different aspects till date. The first area of research focused on mobile app malware classification [9], [10]. However, there were other categories of potentially harmful apps described in the taxonomy in [16] which include rooting apps, privilege escalation apps, etc. Determination of app behavioral patterns could possibly lead to better detection of such potentially harmful apps. Therefore, through this work we have taken a first step towards using system calls made by an app to determine its behavior class.

The second area of research focuses on using NLP techniques. Pandita et. al. [11] were able to achieve an 83% precision and 82% recall in determining why an application uses a permission through NLP techniques. Although a good first step in behavioral analysis, it leaves a lot of room for improvement because their analysis included only 3 popular permissions used by apps. In [12], researchers have attempted to map an app's description from the Google Play Store [4] to its actual behavior. Gorla et.al. [12] provided the following insights. Application vendors hide what their apps do. While Google maintains no standards to avoid deception on a developer's part. This results in developers deceiving users. One such incident occurred in case of the "Brightest Flashlight App", when it collected user location and surreptitiously uploaded the same to an advertising network [3]. The other insight was Android's permission model is flawed and requires lay users to read incomprehensible permission descriptions like "allow access to the device identifier". We want to simplify user decision during permission acquisition.

The final area of research was where taint tracking of sensitive data was used to determine when such information left the mobile device [14]. A system call study can potentially determine such behavior by studying when resources were accessed or manipulated or if network connections were opened and used to send data over such a connection.

III. System Overview

We present our system/process design that we have built for this study in Figure 1. We have five main components in our system: Download module, Annotation module, System call module, Feature generation module and Classification module. The input to our system were search terms for testing app categories. The expected output of the system was behavioral class for an app.

A system call (or syscall) may be defined as the fundamental interface between an application and the Linux kernel [17]. The system calls that are part of Android's kernel distributions have been defined in the class android.system.Os [18]. At the lowest level of the operating system, an app's functionality boils down to the tasks and services it requests the kernel to perform, through system calls. As a result, an app could be monitored by observing the patterns in the system calls it executes.

What does a behavioral class represent? A simple representation of behavioral classes maybe considered as the app category information from Google Play Store. More appropriate category information would be the ones we determined during our behavioral pattern annotation. For example, we found a number of apps that were PDF readers. In order to carry out the annotation, we downloaded all the apps that we could find for this particular behavior category, i.e. PDF readers, on a mobile device. We used the apps and read the app's description on Google Play Store [4] and manually determined the app's

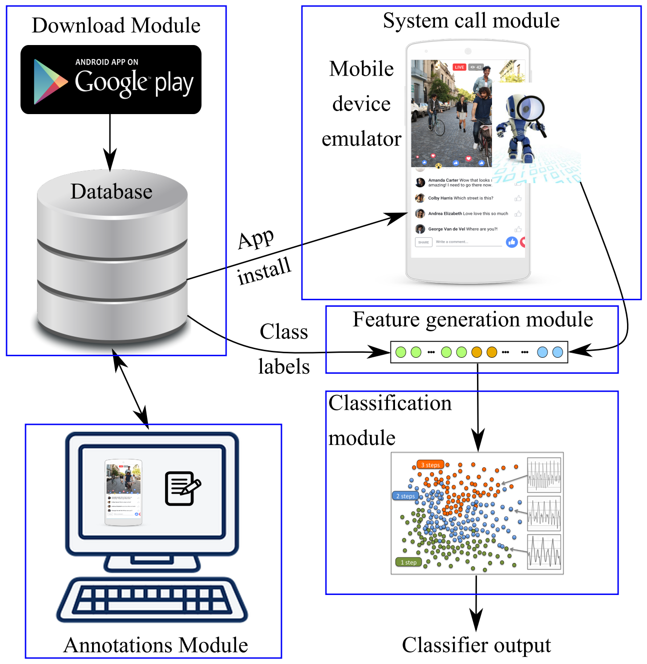

Fig. 1. Design of system built for studying app behavior

primary usage category. Google categorized most of these apps into either Productivity, Books & Reference or Education categories. Such a discrepancy indicates that determination of an app's category was a complicated task in reality.

The `strace` utility enables diagnostics, debugging and instructional user-space monitoring and modifying interactions between processes and the Linux kernel, which include system calls, signal deliveries, and changes of process state. We have used this utility for studying and capturing system calls in form of interactions between user space apps and kernel space programs.

Practical limitations necessitated usage of an emulator for experiments running simulation of user's actions on a mobile device. For that purpose, we used the "UI/Application Exerciser Monkey" [19] (hereafter called Monkey tool). The Monkey tool is a command-line utility that can run on a emulator or mobile device and sends a pseudo-random stream of user events into the system. This utility allowed us to write programs that controlled the Android device or emulator to install a list of apps that were to be tested and exercise all possible UI behavior that a user could trigger.

Once the system calls were captured, we used standard text processing techniques and information retrieval measures like simple term-frequency and tf-idf to generate feature vectors for our classifiers. These feature vectors were then used to run a slew of classifiers using the weka tool [20]. We have presented the classification results in Section V.

IV. System Implementation and Data Setup

Our system was comprised of five system components mentioned in Section III. This section describes the functionality of each of these modules. The emulated Android device we used for our experiments and for the manual annotations step was a Nexus 6, running Android 6.0.1 (Lollipop build December 2015) with Hardware emulation and 1.5GB RAM and 16GB internal storage. Experiments were run on a desktop computer running Ubuntu 14.04 and had a Intel Core-i7 3.4GHz processor, 32GB RAM and 2TB storage space for downloaded apps.

A. Download module

There are several mechanisms for downloading Android apps from the Google Play Store [4]. We used open source code [1] with our system specific modifications for this task. Although our system could be used to study any kind of app and its functional behavioral pattern, for the sake of simplification of experimental evaluation, we started our system with 10 specific search criterion on the Google Play Store [4]. A search on the Play Store [4] could be performed using a simple html GET request with the search term as a url parameter [2]. The task of the download module was used to retrieve both metadata about apps (i.e. app name, developer name, descriptions, Google Play category etc.) and Android executable APK files, to run experiments for apps found through the search results.

B. Annotation module

The annotation module included an interface to read the app description and other meta information and an emulator to install the app and observe its behavior. Based on their observations annotators would ascertain a specific "behavior class" and assign it to the app. We ran our study on 534 Apps from 10 specific keyword patterns. The key word patterns we used include: "alarm clock", "file explorer", "to do list", "scientific calculator", "battery saver", "pdf reader", "video playback", "lunar calendar", "drink recipes", "wifi analyzer". We downloaded 1560 apps found in our search. However, a significant number of these apps, were unusable due to emulator issues (app crashes and incompatibility issues) or because they required some sort of user interaction that could not be automated (for example, profile creation). As a result, we annotated 534 apps.

Table I shows the distribution of apps annotated according to their behavior classes. It is interesting to note that for these 534 apps, Google puts them mostly into Tools and Productivity category as shown in Table II. We can conclude from this observation that not only does Google *NOT* maintain a standardized approach to ensure that a developer explain what their app does, they categorize apps in a very generic fashion. Granular behavior categorization thus remains a motivating challenge for further research.

C. System call module

In the system call module we installed downloaded apps on an Android emulator. We then used the Monkey tool [19] to

[1]CMUChimps Lab: https://github.com/CMUChimpsLab/googleplay-api

[2]URL Prefix: `https://play.google.com/store/search?q=`; Search terms: `pdf readers`; URL Suffix: `&c=apps&hl=en`

TABLE I: Annotated app categories

Annotated behavior class	# apps	%age
Alarm clock	128	23.97
Battery saver	93	17.42
Drink recipes	15	2.81
File explorer	72	13.48
Lunar calendar	12	2.25
Pdf reader	22	4.12
Scientific calculator	61	11.42
To do list	102	19.10
Video playback	5	0.94
Wifi analyzer	24	4.49

TABLE II: Google Play Category

Google Play category	# apps	%age
Tools	265	49.63
Productivity	133	24.91
Lifestyle	48	8.99
Education	14	2.62
Personalization	13	2.43
Books & reference	12	2.25
Music & audio	8	1.50
Entertainment	7	1.31
Communication	6	1.12
Health & fitness	6	1.12
Business	5	0.94
Media & video	4	0.75
Medical	3	0.56
Adventure	2	0.37
Social	2	0.37
Travel & local	2	0.37
Arcade	1	0.19
Libraries & demo	1	0.19
News & magazines	1	0.19
Shopping	1	0.19

simulate a real human using an app and all its functionality. We used the monkey tool to adjust the percentage of "system" key events (like Home, Back, Start Call, End Call, or Volume controls) and maximize coverage of all activities within the app's package. We varied the number of clicks through Monkey between 1000 and 10000 to maximize coverage of "visible" functionality of an app. Throttling was the final option that we used to ensure stability of execution. The final option made sure that we had fewer app crashes. `strace` was used to capture system calls generated by the process of an app. We used the Android Debug Bridge to control the emulator and extract the results of our experiments.

D. Feature generation module

The output of the previous module was a series of system calls made by the app. An excerpt of strace output for an app from the "File Explorer" category is shown below:

```
2966  read(37, ``Android Emulator OpenGL ES
    Trans''..., 65) = 65
2966  read(37, ``A\0\0\0'', 4)              = 4
2966  write(37,
    ``\23'\0\0\24\0\0\0\0\37\0\0\0\0\0\0\0'',
     20) = 20
2966  read(37, ``\34\0\0\0'', 4)            = 4
2966  read(37, ``\344\377\377\377'', 4)     = 4
2966  write(37,
    ``\23'\0\0000\0\0\0\37\0\0\34\0\0''...,
    48) = 48
2966  read(37, ``Google (NVIDIA Corporation)
    \0'', 28) = 28
2966  read(37, ``\347\377\377\377'', 4)     = 4
2966  read(37, ``\31\0\0\0'', 4)            = 4
2966  write(37,
    ``\23'\0\0-\0\214\213\0\0\31\0\0''...,
    45) = 45
2966  read(37, ``OpenGL ES GLSL ES
    1.0.17\0'', 25) = 25
2966  write(37,
    ``\23'\0\0\24\0\0\0\214\213\0\0\0\0\0'',
    20) = 20
```

The first part of each line in `strace` output was the app's process id. After that we have the system call followed by parameters for that particular system call. In our study we collected frequency of system calls made by an app in order to generate features. "Term frequency–inverse document frequency" (tf–idf) [21] is one of the most commonly used term weighting schemes in information retrieval. We compute tf–idf weight vectors using system calls as terms and apps as documents:

$$\boxed{tfidf(t,d,D) = tf(t,d) \times idf(t,D)}$$

where, Term Frequency (TF) was computed as:

$$\boxed{tf(t,d) = 1 + log f_{t,d}}$$

and Inverse Document Frequency (IDF) was computed as:

$$\boxed{idf(t,D) = log\frac{N}{n_t} \Rightarrow idf(t,D) = log\frac{N}{|\{d \epsilon D : t \epsilon d\}|}}$$

Here, 'N' represents the total number of documents in the document set 'D'. 't' represents a term in a specific document 'd'. 'f' represents the frequency of a term 't' in a document 'd'. Finally 'n' represents number of documents 'd' with term 't' in document set 'D'. We generated two sets of feature vectors– one hot vectors and tf–idf weight vectors. We ran the classifiers to train and test on two sets of ground truth labels–annotated class labels and Google Play categories as class labels.

E. Classification module

The classification module consisted of scripts to use the Weka tool [20] and run Support Vector Machine(SVM), Naive

Bayes(NB), Decision tree(J48) and Multilayer Perceptron (MLP) classifiers on our generated feature vectors. For each of these classifiers we used 10 fold cross validation technique and recorded the achieved average F1–measure [22]. F1–measure is the harmonic mean of precision and recall and has a range of [0,1]. For SVM we experimented with RBF and Polynomial kernels.

V. Experimental Evaluation and Discussion

As stated at the beginning of this paper, system calls turned out not to be the best features for behavioral classification. We now take a look at our experimental results. We used four different classifiers through the Weka tool [20]. We used 10 fold cross validation technique for all the classifiers. We present the average F1–measure achieved by each of the classifiers for annotated class labels when using tf–idf weighted feature vectors in Table III. F1–measure for annotated class labels when using 1-hot feature vectors are presented in Table IV. Unfortunately, none of the classifiers recorded a good enough F1–measure for annotated class labels.

TABLE III: Annotated class labels, TF-IDF features

Classifier	F1 score
MLP	0.44
SVM-RBF	0.32
SVM-Poly	0.31
J48	0.27
NB	0.27

TABLE IV: Annotated class labels, one hot features

Classifier	F1 score
J48	0.31
NB	0.27
MLP	0.26
SVM-Poly	0.23
SVM-RBF	0.21

For Google's app category based class labels, average F1–measure achieved by the classifiers were low as well, as shown in Table V, when using tf-idf weighted feature vectors or as shown in Table VI when using one hot feature vectors.

TABLE V: Google class labels, TF-IDF features

Classifier	F1 score
SVM-Poly	0.39
SVM-RBF	0.38
MLP	0.37
J48	0.35
NB	0.14

We observed that when using tf–idf weighted feature vectors we were able to achieve comparatively better classification accuracy as opposed to when using one hot vectors. Intuitively this observation makes sense since tf–idf weights better represent the significance of terms in documents as opposed to simply stating that the document has a certain term. However, a comparison of the classifiers paints a disappointing picture.

TABLE VI: Google class labels, one hot features

Classifier	F1 score
J48	0.39
MLP	0.38
SVM-Poly	0.33
SVM-RBF	0.33
NB	0.09

While MLP performed marginally better than other algorithms, for annotated class labels using tf–idf weighted features, NB did slightly worse than other classifiers for Google class labels. In short, none of the algorithms had an outstanding performance and thus leads to the conclusion that system calls cannot be considered as good features for app behavior classification. We note that this observation was somewhat unexpected, given our understanding of system calls and application functionality correlations, from knowledge of similar methods being used successfully, in the literature [2]. As a result, we came up with two possible explanations for these observed results.

The first was that apps have functionality that belong to multiple behavior classes, for example—an app could have the functionality of social media sharing combined with financial transactions. Take for example WeChat, which has taken over workplaces in China [23]. WeChat combines instant messaging functionality with social media sharing while incorporating functions like ride hailing, buying movie tickets, sending payments, settling utility bills as well as online shopping. Such multi-functional apps, sometimes called "super apps" where apps are trying to become the "only" app on your phone by providing a multitude of functionality. This trend can best be explained by a need to retain a high active-user base, which leads to higher ad-revenue. Ad-revenues understandably are critical for an app's survival today because of the free app economy. As a result, our basic assumption that an app would serve a singular purpose no longer holds true and we need to create coarser functional clusters (i.e. "social media-financial" apps) for behavior analysis.

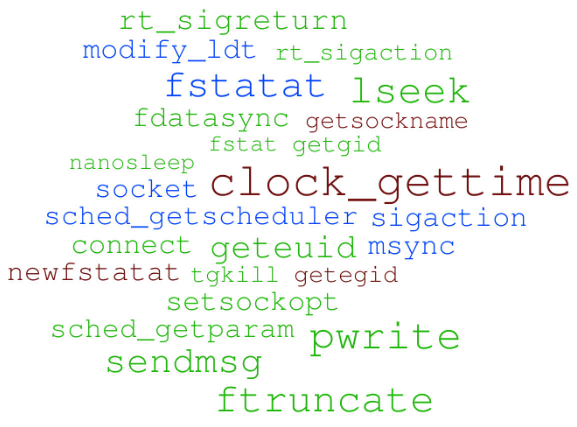

Fig. 2. To do list class

The second related explanation was easier to demonstrate. We observed that since apps are trying to provide a slew of

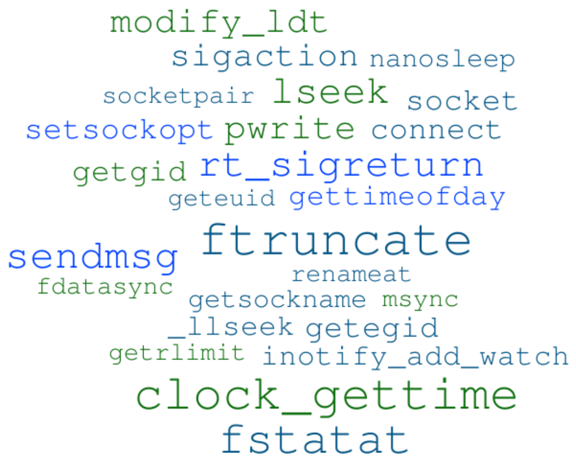

Fig. 3. Scientific calculator class

different functionality, they end up making very similar system calls. In order to investigate this further, we generated the tf-idf word clouds for each of the 10 annotated class of apps. Consider the word clouds for "To Do list" and "Scientific Calculator" shown in Figure 2 and Figure 3. We can clearly see that the "ftruncate", "fstatat" and "clock_gettime" have similar tf-idf weights for both these classes. As a results, these classes were not easily "separable" and despite the expectation that they would different behavioral patterns, were in-fact making similar system calls.

VI. Conclusions and Future Work

In this paper, we wanted to answer the research question: *Can system calls be used to distinguish between how an app "behaves" and its perceived/stated purpose?* We have presented a preliminary study performed with 534 Android apps and showed that using system calls as features was not sufficient when trying to carry out app behavioral classification. Consequently, additional features are required to make such a distinction.

The manual annotation phase of our study required using an app on an emulator. The Android SDK emulator was used since system calls cannot be captured without the `strace` utility, which is unavailable on a real device. However the emulator causes a lot of apps to crash thus complicating our annotation process. There are emulators available for Android, from third party vendors like Genymotion [24], which claim to be stable when compared to the SDK emulator. We are working with such emulators to improve our system call capture stage. In our current study, we used system calls as the only features, which does not result in a good classification accuracy. We acknowledge that the above limitations caused us to work with a restricted data set. We are working to include additional features like app call sequences to improve our system's classification accuracy. We are also exploring the possibilities of using coarser behavior classes, like "social media-financial." Finally, augmenting access control policy decision could be an important goal for app behavior classifiers. We hope to carry out such a study in the future.

VII. Acknowledgment

Support for this work was provided by NSF grants 0910838 and 1228198. We would like to thank Renee Frank and Abhay Kashyap for proofreading the paper.

References

[1] T. Maddox, "Research: 74 percent using or adopting byod," January 2015.
[2] A. P. Kosoresow and S. A. Hofmeyr, "Intrusion detection via system call traces," *IEEE software*, vol. 14, no. 5, p. 35, 1997.
[3] J. M. Kerry O'Brien, Sarah Schroeder, "Ftc approves final order settling charges against flashlight app creator," December 2013.
[4] Google, "Android apps on google play," January 2017.
[5] P. K. Das, D. Ghosh, P. Jagtap, A. Joshi, and T. Finin, "Preserving user privacy and security in context-aware mobile platforms," in *Mobile Application Development, Usability, and Security*. IGI Global, 2016, pp. 166–193.
[6] L. Kagal, T. Finin, and A. Joshi, "A policy language for a pervasive computing environment," in *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*. IEEE, 2003, pp. 63–74.
[7] M. Bokhorst(M66B), "Xprivacy," June 2013.
[8] Google, "Device administration api," January 2017.
[9] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (SP), 2012 IEEE Symposium on*, May 2012, pp. 95–109.
[10] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for android," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 15–26.
[11] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 527–542.
[12] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1025–1035.
[13] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Highly precise taint analysis for android applications," *EC SPRIDE, TU Darmstadt, Tech. Rep*, 2013.
[14] W. Enck, P. Gilbert, B.-G. Chun, L. P., Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010, pp. 1–6.
[15] P. Kumaraguru and L. F. Cranor, "Privacy indexes: a survey of westin's studies," *School of Computer Science, Carnegie Mellon University, Pittsburgh*, 2005.
[16] G. A. Security, "The google android security teams classifications for potentially harmful applications," April 2016.
[17] M. Kerrisk, "syscalls - linux system calls," December 2016.
[18] Google, "Os: Access to low-level system functionality," January 2017.
[19] ——, "Ui/application exerciser monkey," January 2017.
[20] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.
[21] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Commun. ACM*, vol. 18, no. 11, pp. 613–620, Nov. 1975.
[22] C. Van Rijsbergen, "Information retrieval. dept. of computer science, university of glasgow," *URL: citeseer. ist. psu. edu/vanrijsbergen79information. html*, 1979.
[23] Y. Wang, "Tencent's 'super app' wechat is quietly taking over workplaces in china," August 2016.
[24] Genymobile, "Genymotion - fast & easy android emulator," June 2013.