Performance Evaluation on Permission-Based Detection for Android Malware

Chun-Ying Huang, Yi-Ting Tsai, and Chung-Han Hsu

Department of Computer Science and Engineering
National Taiwan Ocean University, Keelung, Taiwan 20224
chuang@ntou.edu.tw, {yttsai,chhsu}@snsl.cs.ntou.edu.tw

Abstract. It is a straightforward idea to detect a harmful mobile application based on the permissions it requests. This study attempts to explore the possibility of detecting malicious applications in Android operating system based on permissions. Compare against previous researches, we collect a relative large number of benign and malicious applications (124,769 and 480, respectively) and conduct experiments based on the collected samples. In addition to the requested and the required permissions, we also extract several easy-to-retrieve features from application packages to help the detection of malicious applications. Four commonly used machine learning algorithms including *AdaBoost*, *Naïve Bayes*, *Decision Tree (C4.5)*, and *Support Vector Machine* are used to evaluate the performance. Experimental results show that a permission-based detector can detect more than 81% of malicious samples. However, due to its precision, we conclude that a permission-based mechanism can be used as a quick filter to identify malicious applications. It still requires a second pass to make complete analysis to a reported malicious application.

Keywords: Android, classification, malware, mobile security, permission.

1 Introduction

An Android application requires several permissions to work. Consequently, an essential step to install an Android application into a mobile device is to allow all permissions requested by the application. Android users must have ever seen a similar screen shot to Figure 1. Before an application is being installed, the system prompts a list of permissions requested by the application and asks the user to confirm the installation. Although Google announced that a security check mechanism is applied to each application uploaded to their market [1], the open design of the Android operating system still allows a user to install any applications downloaded from an untrusted source. Nevertheless, the permission list is still the minimal defense for a user to detect whether an application could be harmful.

Google classifies built-in Android permissions into four categories: normal, dangerous, signature, and signatureOrSystem [2]. Therefore, a straightforward

J.-S. Pan et al. (Eds.): *Advances in Intelligent Systems & Applications*, SIST 21, pp. 111–120.
DOI: 10.1007/978-3-642-35473-1_12

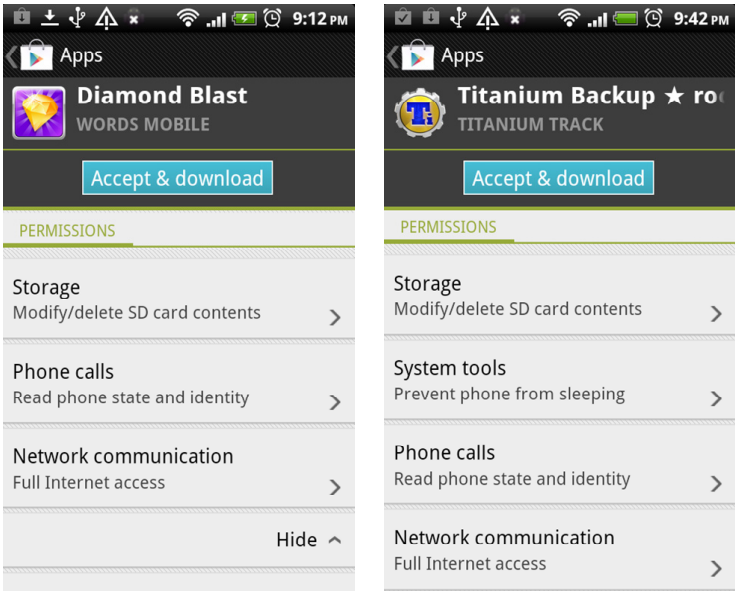


Fig. 1. Example screenshots of asking a user to confirm the installation of applications

idea to determine a harmful application is to check whether it requires a *dangerous permission*. Access permissions to several common activities are classified as dangerous. For example, permissions to read the location of a user (ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION), access bluetooth devices (BLUETOOTH), and access Internet (INTERNET) are all classified as dangerous. However, an application requesting one or more dangerous permissions does not indicate that it is a harmful application. A simple application such as a location-based real-time weather forecast application would need some dangerous permissions such as INTERNET and ACCESS_COARSE_LOCATION. Although Android adopts a coarse-grained permission model to control access to its built-in components, it is not known how good (or bad) it is to detect a malicious application based on permissions or combinations of permissions. It should be noticed that the permissions shown to a user during an installation process are *requested permissions* instead of *required permissions*. The requested permissions are declared by an application developer *manually*. However, not all declared permissions are required by the application. Researchers [3,4] have shown that many developers often declare much more permissions than they actually required. It thereby increases the difficulty on detecting malicious applications based on the permissions.

This study attempts to explore the possibility of detecting malicious applications based on permissions, including both requested and required permissions. Compare against previous researches, a relative large number of benign and malicious applications (124,769 and 480, respectively) were collected and used to conduct the experiments. In addition to the requested and the required

permissions, several easy-to-retrieve features from application packages were extracted as well to help the detection of malicious applications. Four commonly used machine learning algorithms including *AdaBoost*, *Naïve Bayes*, *Decision Tree (C4.5)*, and *Support Vector Machine* are used to explore the possibility.

The remaining of this paper is organized as follows. Section 2 provides a review on several interesting works that analyze permissions of Android applications. Section 3 explains how the features (permissions and other features included in an Android package) are obtained and how the features are labeled (as benign or malicious). Section 4 analyzes the permission requirements of applications and discusses the performance of detectors. Finally, a conclusion and future works are discussed in Section 5.

2 Related Work

A number of researches have introduced and discussed Android permissions. Enck et al. [5] wrote a good introduction on Android's security design in 2009. Basically the Android operating system provides a coarse-grained mandatory access control (MAC). It is able to enforce how applications access components based on granted permissions. Consequently, each Android application must have a list of requested permissions and all these permissions must be granted at the time of installation. The requested permission list is often declared by an application developer manually. Hence, a number of interesting researches are devoted to review how permissions are declared in applications. Barrera et al. [6] analyzed how developers of Android applications use the permissions. They explored and analyzed 1,100 applications using the Self-Organizing Map (SOM) algorithm. They found that although Android has a rich set of permissions, only a small number of these permissions are actively used by developers. Felt et al. [3] studied Android applications to determine whether Android developers follow least privilege with their permission requests. They built a tool and applied it to 940 applications and found that about one-third of evaluated applications are over privileged. They also concluded that developers are trying to follow least privilege but failed due to insufficient API documentation. Johnson et al. [4] developed an architecture that automatically searches for and downloads Android applications from Android Market. With the application, they created a detailed mapping of Android API calls to the required permissions. The idea is similar to [3] but they collected a large number (141,372) of applications to conduct the experiments. They found that the majority of developers are not using the correct permission set. The applications are either over-specify or under-specify their security requirements. Zhou and Jiang [7] systematically characterized 1,260 Android malicious applications from various aspects, including their installation methods, activation mechanisms, and the carried malicious payloads. In addition, they also compared the permission requests of the 1,260 malicious applications against another top free 1,260 benign applications on Android market. The comparison shows that the top 20 frequently requested permissions are similar for both benign and malicious applications.

In addition to analyze permissions, a number of researches tried to detect malicious application using static analysis or dynamic analysis techniques. These techniques are similar to those used to detect traditional malware on desktop personal computers. Besides many well-known signature-based virus scanners, androguard [8] is an open source project that dedicated to detect Android malware. Androguard detect a malicious application or an injected malicious code based on *control flow graph*. A given application package is first disassembled and each identified method in assembly source codes is converted into a formatted string that represents the control flow graph [9] of the method. A number of predefined malware's control flow graphs are then compared against the obtained control flow graph strings to check if they are similar [10] to malware. Schmidt et al. [11] proposed a static analysis solution to detect malicious application based on the output of the *readelf* tool, which contains a list of symbols that involved with an executable. They then differentiate malicious applications from benign ones based on the combinations of system calls used in the executable. Burguera et al. [12] proposed to detect malware using dynamic analysis techniques. They developed a client named *Crowdroid* that is able to monitor Linux kernel system call and report them to a centralized server. Based on the collected dataset, they cluster each dataset using a partition clustering algorithm and hence differentiate between benign and malicious applications. Due to the lack of malware samples, most existing works conduct experiments using self-made malware or a limited number of real malware. It still requires more evidence to prove the effectiveness of these solutions.

3 Feature

For each Android application, we retrieved several selected features from the corresponding application package (APK) file. In addition, we analyzed the source codes of an application, identified real permissions required by the application, and adopted the features for malware detection. The values of selected features are stored as a feature vector, which is represented as a sequence of comma separated values. We enumerate all selected features in the following items. Each item includes the name of a feature, the data type of the feature, and a detailed description about how the features are retrieved.

1. ext_so (*integer*): We list all files found from an APK file and count the number of files with a ".so" extension filename.
2. file_elf (*integer*): We use the UNIX *file* utility to determine the type of each file in an APK file and counts the number of executable and linking format (ELF) files.
3. file_exe (*integer*): Similar to Item 2, but this feature counts only executables.
4. file_so (*integer*): Similar to Item 2, but this feature counts only shared objects.
5. dex.all (*integer*): This feature counts the total number of *required permissions*. As introduced in Section 1, requested permissions and required permissions are different. There is not a file that describes the actual

permissions required by an application. Therefore, it is a must to retrieve the required permissions by analyzing the application from the source-code level. Although Android applications are often written in the Java programming language, here "source codes" are the assembly source codes represented in Jasmin's (dedexer's) syntax. We disassemble byte codes of each Java class file in an APK file into assembly codes using the *baksmali* [13] disassembler. We then identify invoked Android system functions from the assembly codes and look up the required permissions from the permission map table provided by [3]. It should be noticed that currently we only map from function calls to permissions. Although the obtained required permission would be less than all the required permissions, it still improves the performance.

6. dex.normal (*integer*): Google classifies all permissions into four categories, i.e., normal, signature, dangerous, and signatureOrSystem. Among all the 139 built-in permissions[1], 21 permissions are classified as normal, 27 permissions are classified as signatureOrSystem, 35 permissions are classified as signature, and 56 permissions (approximately 40% of all permissions) are classified as dangerous. Similar to Item 5, but this feature counts only the number of permissions that are classified as "normal."
7. dex.sign (*integer*): Similar to Item 5, but this feature counts only the number of permissions that are classified as "signature."
8. dex.dangerous (*integer*): Similar to Item 5, but this feature counts only the number of permissions that are classified as "dangerous."
9. dex.signOrSys (*integer*): Similar to Item 5, but this feature counts only the number of permissions that are classified as "signatureOrSystem."
10. List of all *required permissions* (*boolean*): In addition to count the number of required permissions, we also list the permissions required by an analyzed application. With the retrieved required permission, we convert the permissions into a boolean vector. Suppose the 139 built-in permissions are labeled from 1 to 139 (P_1, P_2, ..., P_{139}), an application that requests P_2 and P_3 would have a boolean vector of values (0, 1, 1, 0, ..., 0). This feature contains 139 boolean values.
11. xml.all (*integer*): This feature counts the number of permissions requested by an application. The requested permissions are retrieved directly from the *AndroidManifest.xml* file that is placed at the root of an APK file. Reading requested permissions from an *AndroidManifest.xml* file is simple. This file can be extracted from an APK file by using the *unzip* tool, convert to a human-readable format using a tool such as *AXMLPrinter2*, and then parsed using the *libxml* library.
12. xml.normal (*integer*): Similar to Item 11, but this feature counts only the number of permissions that are classified as "normal."
13. xml.sign (*integer*): Similar to Item 11, but this feature counts only the number of permissions that are classified as "signature."
14. xml.dangerous (*integer*): Similar to Item 11, but this feature counts only the number of permissions that are classified as "dangerous."

[1] The number is retrieved from the Android 2.3 version (codename: Gingerbread) source tree. Readers can refer to the `frameworks/base/core/res/AndroidManifest.xml`.

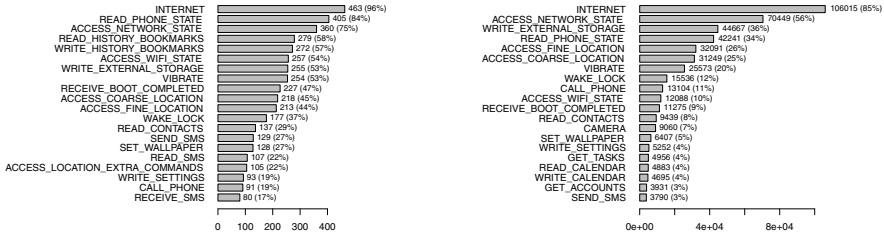
15. xml.signOrSys (*integer*): Similar to Item 11, but this feature counts only the number of permissions that are classified as "signatureOrSystem."
16. List of all *requested permissions*: In addition to count the number of requested permissions, we also list the exact permissions requested by an analyzed application. The format is the same as Item 10. This feature contains 139 boolean values as well.
17. under (*boolean*): This feature is a boolean value to indicate that an application is *under-privileged*. Since the requested permissions listed in an AndroidManifest.xml are declared by the application developer, there are often inconsistencies between the requested permissions and the required permissions. Although a developer should be able to determine which permissions are required by reading the official developer's API reference document, researchers [3] found the documented permission requirements are somewhat different from the actual requirements. Therefore, an application may be *under-privileged* or *over-privileged* depending on how its permission request is declared.

    An under-privileged application means that an application developer requests less permissions than actually the application needs. It could be malfunctioned because of security exceptions raised by the Android operating system when accessing unprivileged system functions. In contrast, an over-privileged application means that an application developer requests more permissions than actually it needs. Although an over-privileged application breaks the ideal least privilege scenario, it does not have any side-effect. Therefore, to prevent an application from being blocked by the Android operating system due to insufficient permissions, a developer often chooses to request more permissions than actually the application needs.
18. ucount (*integer*): This feature counts the number of under-privileged permissions by comparing required permissions against requested permissions. For example, if an application requires INTERNET permission but it does not request the permission, the counter increases by one.
19. over (*boolean*): In contrast to Item 17, this feature is a boolean value to indicate that an application is *over-privileged*.
20. ocount (*integer*): This feature counts the number of over-privileged permissions by comparing required permissions against requested permissions. For example, if an application does not require BLUETOOTH permission but it request the permission, the counter increases by one.

In addition to the selected features, a label BoM is appended at the end of a feature vector to show that the vector belongs to a benign or a malicious application. The value of the BoM contains only malicious and benign. Labeling an application correctly is an important task. We label the obtained feature vectors using three different strategies—*site-based labeling*, *scanner-based labeling*, and *mixed labeling*. Site-based labeling labels an application based on the source we obtain the corresponding APK file. If an APK file is downloaded from Google Play or third party markets, it is labeled as benign. If an APK file is downloaded from a malicious repository, it is labeled as malicious. Scanner-based labeling

```
com.rovio.angrybirds: 2,2,0,2,9,3,6,0,0,0,0,0,1,1,0,0,1,0,1,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,
0,0,0,0,0,0,0,0,0,0,0,6,2,4,0,0,0,0,0,1,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,0,0,0,0,0,1,4,1,1,benign
```

Fig. 2. An example feature vector for the *AngryBird* application retrieved from its Android application package file



(a) Requested Permission (Malicious) **(b)** Requested Permission (Benign)

Fig. 3. Count of requested permissions for both benign and malicious Android applications. The malicious applications are labeled using mixed labeling strategy.

labels an application based on the decision of an anti-virus scanner. Currently we use the open source ClamAV anti-virus software to make the decision. If an APK file is reported to be malicious, it is labeled as malicious. Otherwise it is labeled as benign. Mixed labeling is the union of site-based labeling and scanner-based labeling. If an APK is downloaded from a malicious repository or it is reported as malicious by an anti-virus scanner, it is labeled as malicious. Otherwise, it is labeled as benign. We obtain feature vectors for all the collected 125,249 applications. A complete example of a feature vector for the *AngryBird* game is shown in Figure 2. All the features are retrieved from the *AngryBird*'s APK file. Since we have three different strategies to label the feature vectors, there are three corresponding datasets. The datasets are named by its labeling strategy. We then feed the datasets to machine learning algorithms and evaluate the performance of permission-based detection for malicious Android applications.

4 Result

Before digging into the results of classification, we have a quick look on the most frequently used permissions in Android applications. We plot a similar statistics on top 20 permissions to [7]. Figure 3 shows the top 20 permissions

Table 1. The Performance of Classifiers on Detection of Malicious Applications

	Classifier	TP Rate	FP Rate	Precision	Recall	F-Measure
Dataset #1	AdaBoost	0	0	n/a	0	n/a
Site-based label	Naïve Bayes	0.720000	0.057769	0.019544	0.720000	0.038055
	C4.5 (J48)	0.460000	0.000080	0.901961	0.460000	0.609272
	SVM	0.445000	0.000048	0.936842	0.445000	0.603390
Dataset #2	AdaBoost	0	0.000008	0	0	n/a
Scanner-based label	Naïve Bayes	0.811905	0.076625	0.034424	0.811905	0.066047
	C4.5 (J48)	0.714286	0.000401	0.857143	0.714286	0.779221
	SVM	0.616667	0.000080	0.962825	0.616667	0.751814
Dataset #3	AdaBoost	0	0.000024	0	0	n/a
Mixed label	Naïve Bayes	0.762500	0.086536	0.032787	0.762500	0.062870
	C4.5 (J48)	0.650000	0.000449	0.847826	0.650000	0.735849
	SVM	0.585417	0.000088	0.962329	0.585417	0.727979

requested and required by both malicious and benign applications. Compare our results against their statistics, the top three requested permissions are the same. For malicious applications, the top three requested permissions are INTERNET, READ_PHONE_STATE, and ACCESS_NETWORK_STATE. For benign applications, the top three requested permissions are INTERNET, ACCESS_NETWORK_STATE, and WRITE_EXTERNAL_STORAGE. Although the number of malicious application we evaluated is less than [7], the ranks of requested permissions are similar.

We then use the *Weka* data mining software [14] to classify benign and malicious applications based on permissions. We feed the permission datasets retrieved from the 125,249 applications to four commonly used classifiers. They are *AdaBoost*, *Naïve Bayes*, *C4.5 (J48)*, and *support vector machine (SVM)*. Each classifier builds classification models from the three datasets, distinguishes malicious applications from benign ones based on the models, and then evaluates how good (or bad) it performs. Readers should notice that it is a difficult problem for classifiers because the datasets are *extremely imbalanced datasets*. The ratio of the number of malicious applications and benign applications, in the best case, is 480:124769 (less than 0.004). Finding a malicious application is just like finding a needle in a haystack.

Table 1 shows the performance of each classifier on detection of malicious Android applications. The performance of a classifier is measured using the following metrics: the true positive (TP) rate, the false positive (FP) rate, the precision, the recall rate, and the F-measure. All values range from 0.0 to 1.0. Note that the precision and the F-measure fields of some classifiers are labeled *n/a* in the table. This is because precision is evaluated by $\mathit{true\text{-}positives}/(\mathit{true\text{-}positives} + \mathit{false\text{-}positives})$ but the classifier does not classify any instance into the malicious class. Hence, both *true-positives* and $\mathit{false\text{-}positives}$ are zero. Similarly, the F-measure is evaluated by $2 \cdot (\mathit{precision} \cdot \mathit{recall})/(\mathit{precision} + \mathit{recall})$ and therefore it cannot be evaluated if either precision or recall rate cannot be obtained.

From the table, we also find that the *AdaBoost* classifier does not perform well. It classifies all applications as benign applications. The *Naïve Bayes* classifier does not also perform well because it has a very low precision. The C4.5 (J48) and the SVM would be better choices. They have a much higher precision and the recall rate for the default C4.5 classifier ranges from 0.46 to 0.71. This means that the default C4.5 classifier is possible to identify more than 70% of evaluated malicious applications. For the support vector machine (SVM) classifier, we have tried to optimize it by tuning its *cost* and *gamma* parameter via *cross-validation and grid-search* [15]. Although the optimized performance shows that the recall rate is lower than the C4.5 classifier, the SVM has a very high precision. Based on the result, we are able to choose a classifier to fit different usage different scenarios. If precision is the concern, the C4.5 and the SVM would be good choices. In contrast, if recall rate is the concern, the Naïve Bayes would be a good choice. It is of course that we can combine results from multiple classifiers to get the maximal set of malicious applications. However, we would need a second phase to further examine a detected malicious application.

5 Conclusion and Future Work

Application requested permissions are currently the minimal defense for an Android user to decide whether or not to install an application. This paper explores the possibility of detection malicious Android applications based on permissions and several easy-to-retrieve features from Android application packages. Our large scale experiments show that a single classifier is able to detect about 81% of malicious applications. By combining results from various classifiers, it can be a quick filter to identify more suspicious applications. Although the performance numbers are not perfect, permission-based classifications can be further improved in two directions. First, the retrieval of the required permissions can be further improved by considering permissions relevant to event handling and content accessing. Second, more features can be retrieved by statically analyzing assembly source codes of an application. We believe that permission-based classifications can be a good auxiliary to detect malicious applications.

Acknowledgement. This research was supported in part by National Science Council under the Grants NSC 100-2221-E-019-045 and NSC 101-2219-E-019-001. We would like to thank the anonymous reviewers for their valuable and helpful comments.

References

1. Lockheimer, H.: Android and security. Official Google Mobile Blog (February 2012), http://googlemobile.blogspot.tw/2012/02/android-and-security.html
2. <permission>. Android Developer - API Guides - Android Manifest, http://developer.android.com/guide/topics/manifest/permission-element.html

3. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 627–638 (2011)
4. Johnson, R., Wang, Z., Gagnon, C., Stavrou, A.: Analysis android applications' permissions. In: Proceedings of the 6th International Conference on Software Security and Reliability (2012)
5. Enck, W., Ongtang, M., McDaniel, P.: Understanding android security. IEEE Security and Privacy 7(1), 50–57 (2009)
6. Barrera, D., Kayacik, H.G., van Oorschot, P.C., Somayaji, A.: A methodology for empirical analysis of permission-based security models and its application to android. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 73–84 (2010)
7. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: Proceedings of the 33rd IEEE Symposium on Security and Privacy, pp. 95–109 (2012)
8. Desnos, A.: androguard - reverse engineering, malware and goodware analysis of android applications ... and more (ninja !). Googld Project Hosting, `http://code.google.com/p/androguard/`
9. Cesare, S., Xiang, Y.: Classification of malware using structured control flow. In: Proceedings of the 8th Australasian Symposium on Parallel and Distributed Computing (2010)
10. Pouik, G0rfi3ld: Similarities for fun & profit. Phrack #68 (April 2012), `http://www.phrack.org/issues.html?issue=68&id=15#article`
11. Schmidt, A.D., Bye, R., Schmidt, H.G., Clausen, J., Kiraz, O., Yuksel, K.A., Camtepe, S.A., Albayrak, S.: Static analysis of executables for collaborative malware detection on android. In: Proceedings of IEEE International Conference on Communications (2009)
12. 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 (2011)
13. Freke, J.: smali - an assembler/disassembler for android's dex format. Google Project Hosting, `http://code.google.com/p/smali/`
14. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: an update. SIGKDD Explorations Newsletter 11(1), 10–18 (2009)
15. Hsu, C.W., Chang, C.C., Lin, C.J.: A practical guide to support vector classification. Technical report, Department of Computer Science and Information Engineering, National Taiwan University (2009), `http://www.csie.ntu.edu.tw/%7ecjlin/libsvm/`