Android Malware Detection based on Software Complexity Metrics

Mykola Protsenko and Tilo Müller

Department of Computer Science
Friedrich-Alexander-Universität Erlangen-Nürnberg
mykola.protsenko@fau.de, tilo.mueller@cs.fau.de

Abstract. In this paper, we propose a new approach for the static detection of Android malware by means of machine learning that is based on software complexity metrics, such as *McCabe's Cyclomatic Complexity* and the *Chidamber and Kemerer Metrics Suite*. The practical evaluation of our approach, involving 20,703 benign and 11,444 malicious apps, witnesses a high classification quality of our proposed method, and we assess its resilience against common obfuscation transformations. With respect to our large-scale test set of more than 32,000 apps, we show a true positive rate of up to 93% and a false positive rate of 0.5% for unobfuscated malware samples. For obfuscated malware samples, however, we register a significant drop of the true positive rate, whereas permission-based classification schemes are immune against such program transformations. According to these results, we advocate for our new method to be a useful detector for samples within a malware family sharing functionality and source code. Our approach is more conservative than permission-based classifications, and might hence be more suitable for an automated weighting of Android apps, e.g., by the Google Bouncer.

1 Introduction

According to a recent security report [1], Android still remains the most popular platform for malware writers. About 99% of new mobile malware samples found in 2013 are targeting Android. Despite the attention this threat has drawn in both academia and industry, the explosive growth of known Android malware between 2011 and 2012 could not effectively be restricted. Quite the contrary, the number of Android malware samples was reported to have increased four times between 2012 and 2013, while the number of malware families increased by only 69% over the same period of time [2]. These numbers suggest that new malware samples most often belong to existing malware families, supporting the theory of broad code reuse among malware authors, and hence, reasoning our approach to classify malware based on software metrics. Moreover, these statistics confirm considerable flaws in current malware detection systems, e.g., inside the Google Bouncer. As a consequence, we must keep looking for efficient alternatives to detect Android malware without rejecting legitimate apps.

1.1 Contributions

In this paper, we address the issue of static malware detection by proposing a new approach that utilizes machine learning applied on attributes which are based on software complexity metrics. Complexity metrics can be found in the classic literature for software engineering and are known as *McCabe's Cyclomatic Complexity* [3] and the *Chidamber and Kemerer Metrics Suite* [4], for example (see Sect. 2.1). Our selected set of metrics comprises control- and data-flow metrics as well as object-oriented design metrics. To assess the effectiveness of our proposed method, we perform a large-scale evaluation and compare it to Android malware classification based on permissions. As permission-based malware classification is well-known in the literature [5], and has already been applied in practice by sandboxes, we use its detection rate as a reference value.

In our first scenario, we involve more than 32,000 apps, including over 11,000 malicious apps, and demonstrate that the detection rate of our method is more accurate than its permission-based counterpart. For example, the true positive rate of our method reaches up to 93%, just like the permission-based approach, but its overall AUC value [6] is higher due to a better false positive rate, namely 0.5% rather than 2.5%. In a second scenario, which involves over 30,000 apps, we utilize strong obfuscation transformations for changing the code structure of malware samples in an automated fashion. This obfuscation step is based on *PANDORA* [7], a transformation system for Android bytecode without requiring the source code of an app. In consequence of this obfuscation, the metrics-based detection experiences a decrease of its accuracy, whereas the permission-based approach is independent from an app's internal structure. For example, the AUC value of our approach decreases to 0.95 for obfuscated malware, while the permission-based approach remains 0.98.

According to these results, we advocate for our new method to be a useful detector for "refurbished" malware in the first place, i.e., for malware samples within a family that shares functionality and source code. If the detection of shared code is intentionally destroyed by obfuscation, or if new malware families emerge, traditional permission-based methods outperform our approach. However, permission-based methods often misclassify social media apps and those that require an immense set of privacy-related permissions. With respect to these apps, our approach is more conservative and could hence be more practical for weighting systems like the Google Bouncer.

1.2 Background and Related Work

The classification of malware based on machine learning has a long history on Windows. In 2006, Kolter and Maloof [8] have applied machine learning on features such as *n-grams* of code bytes, i.e., sequences of n bytes of binary code. Since the number of distinct n-grams can be quite large, they applied an information gain attribute ranking to select most relevant n-grams. Their practical evaluation involved more than 3,500 benign and malicious executables and indicated a detection performance with a true positive rate of 0.98 and a

false positive rate of 0.05. In 2013, Kong and Yan [9] proposed an automated classification of Windows malware based on function call graphs, extended with additional features such as API calls and I/O operations.

On Android, recently proposed malware classification based on static features utilizes attributes that can easily be extracted, such as permissions. *DroidMat*, presented by Dong-Jie et al. [10] in 2012, consults permissions, intents, inter-component communication, and API calls to distinguish malicious apps from benign ones. The detection performance was evaluated on a data set of 1,500 benign and 238 malicious apps and compared with the Androguard risk ranking tool with respect to detection metrics like the accuracy. In 2013, Sanz et al. [5, 11, 12] performed an evaluation of machine learning approaches based on such static app properties as permissions [5], string constants [12], and uses-feature tags of Android manifest files [11]. Their evaluation involved two data sets, one with 357 benign and 249 malicious apps, the other with 333 benign and 333 malicious apps. In 2014, Arp et al. [13] presented *DREBIN*, an on-device malware detection tool utilizing machine learning based on features like requested hardware components, permissions, names of app components, intents, and API calls. The large-scale evaluation on the data set with nearly 130,000 apps demonstrated a detection performance of 94% with a false positive rate of 1%, outperforming the number of competing anti-virus scanners.

Besides static malware detection, machine learning is often used in combination with dynamic malware analysis. In 2011, the *Crowdroid* system by Burguera et al. [14] was proposed to perform the detection of malicious apps based on their runtime behavior, which is submitted to a central server rather than being processed on the device. This scheme aims to improve the detection performance by analyzing behavior traces collected from multiple users. In 2012, Shabtai et al. [15] proposed a behavioral based system named *Andromaly*, which also employs machine learning for the detection of malware based on dynamic events that are collected at an app's runtime.

2 Attribute Sets for Machine Learning

In this section, we introduce software complexity metrics known from the software engineering literature and define which of these metrics we pick for our attribute set (Sect. 2.1). Moreover, as our practical evaluation is based on the comparison of two attribute sets, we discuss Android-specific attributes such as permissions (Sect. 2.2).

2.1 Software Complexity Metrics

Software complexity metrics were traditionally used to ensure the maintainability and testability of software projects, and to identify code parts with potentially high bug density, in the field of software engineering. Our set of selected metrics reflects the complexity of a program's control flow, data flow, and object-oriented design (OOD). These metrics turned out to be also useful in the field of malware

classification. In the following, we describe the selected metrics in more detail. To employ them in our detection system, we implemented their computation on top of the *SOOT* optimization and analysis framework [16].

Lines of Code The first and the simplest metric we use is the number of Dalvik instructions, which we denote as the number of lines of code (LOC).

McCabe's Cyclomatic Complexity One of the oldest and yet still most widely used metrics is the cyclomatic complexity first introduced by McCabe in 1976 [3]. This complexity measure is based on the cyclomatic number of a function's control flow graph (CFG), which corresponds to the number of the linearly independent paths in the graph. Grounding on McCabe's definition, we compute the control flow complexity of a function as $v = e - n + r + 1$, with e, n, and r being the number of edges, nodes, and return nodes of the control flow graph, respectively.

The Dependency Degree As a measure for a function's data flow complexity, we use the dependency degree metric proposed by Beyer and Fararooy in 2010 [17]. This metric incorporates dependencies between the instructions using local variables and their defining statements. For a given CFG, its dependency graph $S_G = (B, E)$ is built by B, which is defined as the node set corresponding to a function's instruction set, and E, which is the set of directed edges that connect the instruction nodes with other instructions they depend on. The dependency degree of one instruction is defined as the degree of its corresponding node. The dependency degree of a whole function is defined as the sum of the dependency degrees of all its instructions, i.e., the total number of edges in a function's dependency graph.

The Chidamber and Kemerer Metrics Suite The two previously described metrics both measure the complexity of a single function. The complexity of an app's object-oriented design can be evaluated with the metrics suite proposed by Chidamber and Kemerer in 1994 [4]. The six class complexity metrics of this suite are defined as follows:

- *Weighted Methods per Class (WMC)* is defined as a sum of all methods complexity weights. For the sake of simplicity, we assign each method a weight 1 yielding the *WMC* the total number of methods in a given class.
- *Depth of Inheritance Tree (DIT)* is the classes depth in the inheritance tree starting from the root node corresponding to `java.lang.Object`.
- *Number of Children (NOC)* counts all direct subclasses of a given class.
- *Coupling Between the Object classes (CBO)* counts all classes a given one is coupled to. Two classes are considered coupled, if one calls another's methods or uses its instance variables.
- *Response set For a Class (RFC)* is the sum of methods that is declared in a class and the methods called from those.

– *Lack of Cohesion in Methods (LCOM)*: For class methods $M_1, M_2, ..., M_n$, let I_j be the set of the instance variables used by method M_j, and define $P = \{(I_i, I_j) | I_i \bigcap I_j = \emptyset\}$ as well as $Q = \{(I_i, I_j) | I_i \bigcap I_j \neq \emptyset\}$. The LCOM metric is then computed as $LCOM = max\{|P| - |Q|, 0\}$.

Aggregation Since all previously described metrics measure either the complexity of a single method or that of a single class, additional processing is required to convert those into whole-app attributes. In a first step, we convert method-level metrics into class-level metrics by aggregating the metrics of all classes methods with the following six functions: minimum (*min*), maximum (*max*), sum (*sum*), average (*avg*), median (*med*), and variance (*var*). In a second step, the class-level metrics, including those resulting from the first step, are aggregated in the same way for the whole app. For example, the app attribute `Cyclomatic.var.max` denotes the maximum variance of the cyclomatic complexity among all classes. According to these aggregation rules, for the three method-level and six class-level metrics described in the previous paragraphs, we obtain $3 \cdot 6 \cdot 6 + 6 \cdot 6 = 144$ complexity attributes in total.

2.2 Android-Specific Attributes

Our second attribute set is given by Android-specific attributes such as permissions. We investigated Android-specific features to compare them with software complexity metrics regarding their usefulness for malware detection. Note that in general, permissions requested in a manifest file often differ from the set of actually used permissions. Moreover, the number of available permissions varies with the Android API level [18]. In our study, we utilized the *Androguard* tool by Desnos and Gueguen [19], which supports a set of 199 permissions.

Aside from Android permissions, we also extracted eight features that are mostly specific to Android apps and can additionally be extracted by the Androguard tool, including the number of the app components, i.e., *Activities*, *Services*, *Broadcast Receivers* and *Service Providers*, as well as the presence of *native*, *dynamic*, and *reflective code*, and the ASCII obfuscation of *string constants*. Taking into account 144 complexity metrics, as described above, 199 permissions, and these eight attributes, gives us a total number of 351 app attributes serving as a basis for our evaluation. As explained in Sect. 4.3, however, the latter eight attributes were inappropriate for classification and discarded.

3 Obfuscation Transformations

Recent studies [7, 20] confirm the low performance of commercial anti-virus products for Android in face of obfuscation and program transformations. To overcome such flaws in the future, new malware detection systems must compete with obfuscation. To evaluate the resilience of our detection system against common program transformations, we have applied various code obfuscation techniques to a set of malware samples. The obfuscation was performed by

means of the *PANDORA* framework proposed by Protsenko and Müller [7]. The provided transformation set is able to perform significant changes to an app's bytecode without requiring its source code. These transformations strongly affect the complexity metrics described above, without affecting its Android-specific attributes like permissions. In Section 3.1 and 3.2, we give a brief description of all transformations we applied.

3.1 Data and Control Flow Obfuscation

The first group of obfuscation techniques we applied aims to disguise the usage of local variables by obfuscating the control and data flow of program methods. These transformations tend to increase the values of the method-level metrics such as the cyclomatic complexity and the dependency degree.

- *String Encryption*: We encrypt string constants by using the *Vigenere* encryption algorithm as well as a modification of the *Caesar* cipher with the key sequence generated the *Linear Congruential Method*. The decryption code is inserted after each string constant.
- *Integer Encoding*: Integer variables are obfuscated by splitting them into a quotient and a remainder for a constant divisor, and by applying a linear transformation to it, i.e., by multiplying a constant value and adding another constant value.
- *Array Index Shift*: The access to array elements is obfuscated by shifting their index values.
- *Locals Composition*: We unite some of the local variables of the same type into arrays and maps with random integer, character, and string keys.

3.2 Object-Oriented Design Obfuscation

The following obfuscation transformations modify and increase the complexity of an app's object-oriented design.

- *Encapsulate Field*: For a given instance or class variable, we create getter and setter methods and modify all usages and definitions of this variable to utilize these auxiliary methods.
- *Move Fields and Methods*: Both static and non-static methods and fields are occasionally moved from one class to another.
- *Merge Methods*: Two methods declared in the same class and having the same return type can be merged in one method, which has a combined parameter list plus one additional parameter to decide which of the code sequences is to be executed.
- *Drop Modifiers*: For classes, methods and fields, we discard access restriction modifiers, i.e., `private`, `protected`, and `final`, which allows for more obfuscation transformations.
- *Extract Method*: For methods with a signature that cannot be changed, e.g., app entry points, we outline method bodies to new methods. This enables other transformations to be applied on those methods, too.

4 Evaluation

In this section, we first describe the data sets and evaluation criteria we used to test our metrics-based classification scheme. We then summarize the results of two test scenarios, one with plain Android apps and the other with obfuscated apps, as described above. For the practical classification, we utilized the *WEKA* machine learning framework by Hall et al. [21]. WEKA is a collection of popular machine learning algorithms that can be applied to a given data set without the need to implement algorithms from-scratch.

4.1 Data sets of Benign and Malicious Android Apps

The data sets we used during our evaluation were composed of Android apps collected for the Mobile-Sandbox by Spreitzenbarth et al. [22]. These apps were already classified into benign and malicious samples. Out of over 136,000 available apps from Google's official Play Store, and out of over 40,000 malicious samples identified by *VirusTotal*, representing 192 malware families, we randomly selected 32,147 distinct apps. In detail, we selected 20,703 benign apps and 11,444 malicious apps from 66 families, with at least 10 samples per family. We grouped the selected samples into the following data sets, each containing only distinct apps in their original or obfuscated form:

- *Dataset 1*: 32,147 apps; 20,703 benign and 11,444 malicious.
- *Dataset 2*: 16,104 apps; 10,380 benign and 5,724 malicious.
- *Dataset 3*: 14,221 apps; 10,323 benign and 3,898 obfuscated malicious.

Note that for each malicious sample from *Dataset 3*, there are samples in *Dataset 2* representing the same malware family.

4.2 Evaluation Criteria

During our evaluation, we have employed the usual metrics for classifying the quality of machine learning algorithms [23], outlined in the following:

- *True Positive Rate*: The ratio of correctly detected malicious samples.
- *False Positive Rate*: The ratio of misclassified benign samples.
- *Accuracy (ACC)*: The number of correctly classified samples.
- *Area under the ROC Curve (AUC)*: The probability that a randomly chosen malicious sample will be correctly classified [6].

AUC can be considered as the summary metric reflecting the classification ability of an algorithm. Hosmer et al. [23] propose the following guidelines for assessing the classification quality by the AUC value: For $AUC \geq 0.9$ they refer to an *outstanding* discrimination, $0.8 \geq AUC < 0.9$ yields an *excellent* discrimination, and for $0.7 \geq AUC < 0.8$ the discrimination is considered *acceptable*. AUC values below 0.7 witness poor practical usefulness of the classifier.

Algorithm	AUC	ACC	TPR	FPR
RandomForest -I 100	0.993	97.3186	0.934	0.005
RotationForest	0.988	96.6933	0.924	0.010
Bagging	0.985	95.3339	0.902	0.019
Decorate	0.977	94.7647	0.936	0.046
DTNB	0.970	93.1595	0.847	0.022
IBk -K 3	0.967	92.7303	0.918	0.068
PART	0.958	91.5855	0.863	0.055

(a) Complexity Metrics

Algorithm	AUC	ACC	TPR	FPR
RandomForest -I 50	0.969	93.9590	0.878	0.026
IBk -K 1	0.968	93.8812	0.880	0.029
Bagging	0.965	93.5173	0.875	0.031
DTNB	0.965	93.4831	0.858	0.023
RandomTree	0.964	93.8937	0.878	0.028
PART	0.962	93.7941	0.875	0.027
RotationForest	0.958	93.8346	0.877	0.028

(b) Permissions

Table 1: No Obfuscation, 25 attributes.

4.3 Attribute Ranking

As described in Sect 2, the attribute set of our evaluation includes 144 complexity metrics, 199 permissions, and eight other Android-specific features. For the resulting total number of app attributes, namely 351, we do not expect all attributes to be equally useful for the distinction of malicious and benign apps. For instance, some permissions do not occur in any app of our test sets, such as `MANAGE_USB` and `WRITE_CALL_LOG`. Additionally, some of the aggregated complexity metrics are expected to have high correlation, and hence induce attributes that can be dropped without significant loss of the prediction quality.

To reduce the attributes to a small subset of the most relevant ones, we follow the example of Kolter and Maloof [8] and perform attribute ranking by means of the information gain. Note that the result of this ranking depends on the training set, so we obtained different ranking results for our two different scenarios. In both cases, however, the top-ranked permissions are `SEND_SMS`, `READ_PHONE_STATE`, `RECEIVE_BOOT_COMPLETED`, and `READ_SMS`. The top Android-specific features are represented by *number of services* and *broadcast receivers*.

Also note that in both scenarios, the complexity metrics clearly dominate in the Top-100 ranking, leaving place for only two Android-specific features, and four or two permissions in case of obfuscation disabled or enabled, respectively. Therefore, due to their low ranking, we decided to exclude Android-specific features other than permissions from further evaluation. From the remaining attributes, we analyzed two distinct attribute sets: complexity metrics and permissions. This decision was also supported by our goal to maintain comparability of the metrics with previous permission-based classification schemes from Sanz et al. [5].

4.4 Scenario 1: No Obfuscation

In the first evaluation scenario, we measure the qualities of metrics-based classification in comparison to permission-based classification without obfuscating apps. For this purpose, we performed a ten-fold cross-validation on *Testset 1* with different machine learning algorithms. The classification results for the top performing algorithms from the WEKA framework [21], namely *RandomForest*, *RotationForest*, *Bagging*, *Decorate*, *DTNB*, *IBk*, and *PART*, are summarized in Tab. 1 for the Top-25 attributes, and in Tab. 2 for the Top-50 attributes.

For both Top-25 and Top-50 selected attributes, the results demonstrated by the permission-based detection were slightly outperformed by our new approach

Algorithm	AUC	ACC	TPR	FPR
RandomForest -I 100	0.992	97.1164	0.930	0.006
RotationForest	0.988	96.5596	0.925	0.012
Decorate	0.986	96.4196	0.931	0.017
Bagging	0.986	95.4273	0.903	0.017
IBk -K 3	0.972	93.2995	0.929	0.065
DTNB	0.969	93.2498	0.847	0.020
PART	0.958	92.6867	0.902	0.059

(a) Complexity Metrics

Algorithm	AUC	ACC	TPR	FPR
RandomForest -I 100	0.984	95.7725	0.930	0.027
IBk -K 3	0.983	95.0135	0.927	0.037
Bagging	0.978	94.7522	0.912	0.033
DTNB	0.976	94.8829	0.897	0.023
RotationForest	0.974	95.2873	0.914	0.026
PART	0.973	95.2219	0.922	0.031
Decorate	0.973	94.7118	0.913	0.034

(b) Permissions

Table 2: No Obfuscation, 50 attributes.

Algorithm	AUC	ACC	TPR	FPR
RandomForest -I 50	0.867	72.2242	0.013	0.010
Bagging	0.808	71.9640	0.049	0.027
NaiveBayes	0.796	84.9589	0.758	0.116
DTNB	0.784	73.5532	0.144	0.041
RotationForest	0.691	71.8866	0.014	0.015
IBk -K 3	0.659	70.5858	0.146	0.083
Decorate	0.644	71.6476	0.068	0.039

(a) Complexity Metrics

Algorithm	AUC	ACC	TPR	FPR
RandomForest -I 50	0.963	94.4378	0.867	0.026
IBk -K 1	0.962	94.2761	0.870	0.030
Bagging	0.959	93.8190	0.862	0.033
DTNB	0.958	94.1706	0.847	0.023
PART	0.956	94.0721	0.860	0.029
RotationForest	0.947	94.0089	0.860	0.030
Decorate	0.941	93.7768	0.861	0.033

(b) Permissions

Table 3: Obfuscation enabled, 25 attributes.

based on complexity metrics. Among all attribute sets and classification algorithms, the overall leadership belongs to *RandomForest* with 100 trees trained on the Top-25 complexity metric-attributes with an outstanding AUC value of 0.993, and the true positive and false positive rates of 93.5% and 0.5% respectively. It is worth noting, that an increase from 25 to 50 attributes resulted in a slightly lower performance for the metrics-based scheme, whereas for the permission-based scheme the opposite was the case.

The result of our experiment for the permission-based classification is consistent with the results obtained by Sanz et al. [5], although they obtained slightly weaker results. Moreover, Sanz et al. did not perform attribute ranking but utilized complete permission set. In terms of AUC, the best algorithm in their evaluation was *RandomForest* with 10 trees, showing the AUC, true positive, and false positive rates of 0.92, 0.91, and 0.19, respectively.

4.5 Scenario 2: Obfuscation Enabled

In the second evaluation scenario, we tested the classification effectiveness of both approaches with the presence of obfuscated apps. Since we decided to put obfuscated samples only in the test set, but not in the training set, we had to refrain from employing cross-validation in this scenario. Instead, we used *Dataset 2* as a training set, and *Dataset 3* as an evaluation set. The results of our evaluation for the complexity metrics and permission-based approaches with the Top-25 and Top-50 attributes, respectively, are given in Tab. 3 and Tab. 4.

As expected, the classification performance of the permissions-based machine learning keeps the high level showed in the previous scenario, since obfuscation transformations do not affect app permissions. The complexity metrics-based classification, on the contrary, shows a significant drop of its detection quality. In case of Top-25 attributes, the results are not satisfactory at all, whereas for

Algorithm	AUC	ACC	TPR	FPR
SimpleLogistic	0.947	92.8908	0.885	0.055
RandomForest -I 100	0.940	75.5995	0.129	0.007
Bagging	0.921	84.9940	0.519	0.025
RotationForest	0.836	73.7993	0.083	0.015
LogitBoost	0.812	80.2686	0.461	0.068
Dagging	0.753	75.9581	0.144	0.008
Decorate	0.793	73.736	0.140	0.037

(a) Complexity Metrics

Algorithm	AUC	ACC	TPR	FPR
RandomForest -I 100	0.981	95.5629	0.916	0.029
IBk -K 1	0.979	95.2746	0.919	0.035
Bagging	0.975	94.6417	0.888	0.031
DTNB	0.971	95.2394	0.877	0.019
RandomTree	0.967	94.9300	0.915	0.038
Decorate	0.967	94.5292	0.894	0.035
PART	0.965	94.8878	0.909	0.036

(b) Permissions

Table 4: Obfuscation enabled, 50 attributes.

	AUC	ACC	TPR	FPR
No Obfuscation	0.965	93.4955	0.873	0.031
Obfuscation Enabled	0.838	73.6165	0.061	0.009

(a) 25 Attributes

	AUC	ACC	TPR	FPR
No Obfuscation	0.990	96.2983	0.912	0.009
Obfuscation Enabled	0.965	92.3845	0.754	0.012

(b) 50 Attributes

Table 5: The Voting Approach, RandomForest -I 100 and SimpleLogistic.

the Top-50 attributes, we can distinguish the *SimpleLogistic* classifier with more than acceptable values of AUC, and true positive and false positive rates, namely 0.947, 92.9%, and 0.055, respectively.

Since according to our results, the best performance for the Top-50 complexity metrics attributes was shown by *RandomForest* with 100 trees for no obfuscation and by *SimpleLogistic* in case of enabled obfuscation, we have tried to combine those classifiers in order to achieve good performance in both evaluation scenarios. For this purpose, we have used the *Voting* approach, which allows the combination of single classifiers. The evaluation results for the combination of the *RandomForest* with 100 trees and *SimpleLogistic* classifier are presented in Tab. 5.

According to the results of the voting-based approach, we were able to improve the detection for obfuscated apps without significantly loosing quality in the non-obfuscation scenario. The classification quality, however, is still only acceptable in case we use the Top-50 attributes. At this point, we want to emphasize that although the results showed by the permission-based detection outperform the proposed complexity metrics-approach, their results can still be considered high. Furthermore, the false positive rate for obfuscated samples is at most 1.2% and hence, consistently lower than the false positive rate of the permission-based approach. As a consequence, metrics-based classification can be considered more conservative than permission-based classification.

To substantiate our assertion, we have additionally investigated the classification of twelve popular social media and messaging apps, including Facebook, Google+, Skype, ChatOn, and more. This kind of apps is particularly interesting, as they are known for their extensive use of privacy-related permissions and regularly head the Play Store list of popular apps. Whereas the permission-based approach misclassified three out of twelve social media apps, which corresponds to a false positive rate of 25%, the metrics-based approach did not misclassify any of these apps.

5 Conclusion and Future Work

In this paper, we proposed a novel machine learning approach for static malware detection on Android that is based on software complexity metrics rather than permissions. Our large-scale evaluation, comprising data sets of more than 32,000 apps in total, has indicated a high detection quality for unobfuscated malware samples which outperforms the classic permission-based approach. In our evaluation scenario involving obfuscation transformations, however, the metrics-based detection has a lower true positive rate than the permission-based detection. As an advantage, also the false positive rate is lower and hence, we emphasize the usefulness of our method to detect new or refactored samples of known malware families. The metrics-based classification can be considered for conservative app weighting in automated analysis systems, such as the Google Bouncer. To substantiate our line of reasoning, we investigated twelve popular social media apps showing a high false positive rate for permission-based classification but none for metrics-based classification. Investigating this effect in more detail, and limiting our false positive rate further, remains an important subject of future work.

Acknowledgments

The research leading to these results was supported by the "Bavarian State Ministry of Education, Science and the Arts" as part of the FORSEC research association. Furthermore, we want to thank Johannes Götzfried and Dominik Maier for proofreading our paper and giving us valuable hints for improving it. A special thanks goes to Michael Spreitzenbarth for giving us access to a large set of benign and malicious Android apps.

References

1. Cisco Systems Inc.: Cisco 2014 Annual Security Report. `https://www.cisco.com/web/offer/gist_ty2_asset/Cisco_2014_ASR.pdf` Accessed: 2014-03-18.
2. Bartlomiej Uscilowski: Symantec Security Response (Mobile Adware and Malware Analysis). `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/madware_and_malware_analysis.pdf` Accessed: 2014-03-18.
3. McCabe, T.J.: A Complexity Measure. IEEE Transactions on Software Engineering **SE-2**(4) (December 1976) 308–320
4. Chidamber, S. R. and Kemerer, C. F.: A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering **20**(6) (June 1994) 476–493
5. Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Bringas, P., lvarez, G.: PUMA: Permission Usage to Detect Malware in Android. In Herrero, ., Snel, V., Abraham, A., Zelinka, I., Baruque, B., Quintin, H., Calvo, J.L., Sedano, J., Corchado, E., eds.: International Joint Conference CISIS'12-ICEUTE'12-SOCO'12 Special Sessions. Volume 189 of Advances in Intelligent Systems and Computing. Springer (2013)
6. Hanley, J.A., McNeil, B.J.: The meaning and use of the area under a receiver operating characteristic (ROC) curve. Radiology **143**(1) (1982) 29–36
7. Protsenko, M., Müller, T.: PANDORA Applies Non-Deterministic Obfuscation Randomly to Android. In Fernando C. Osorio, ed.: 8th International Conference on Malicious and Unwanted Software (Malware '13). (October 2013)

8. Kolter, J.Z., Maloof, M.A.: Learning to Detect and Classify Malicious Executables in the Wild. Journal of Machine Learning Research **7** (December 2006) 2721–2744
9. Kong, D., Yan, G.: Discriminant Malware Distance Learning on Structural Information for Automated Malware Classification. In: Proceedings of the International Conference on Measurement and Modeling of Computer Systems. SIGMETRICS '13, New York, NY, USA, ACM (2013) 347–348
10. Wu, D.J., Mao, C.H., Wei, T.E., Lee, H.M., Wu, K.P.: DroidMat: Android Malware Detection through Manifest and API Calls Tracing. In: Seventh Asia Joint Conference on Information Security (Asia JCIS '12), Tokyo, Japan (August 2012)
11. Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Nieves, J., Bringas, P.G., lvarez Maran, G.: MAMA: Manifest Analysis for Malware detection in Android. Cybernetics and Systems **44**(6-7) (2013) 469–488
12. Sanz, B., Santos, I., Nieves, J., Laorden, C., Alonso-Gonzalez, I., Bringas, P.: MADS: Malicious Android Applications Detection through String Analysis. In Lopez, J., Huang, X., Sandhu, R., eds.: Network and System Security. Volume 7873 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 178–191
13. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In: Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), San Diego, CA (February 2014)
14. 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, New York, NY, USA (2011) 15–26
15. Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., Weiss, Y.: Andromaly: A Behavioral Malware Detection Framework for Android Devices. Journal of Intelligent Information Systems **38**(1) (2012) 161–190
16. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - A Java Bytecode Optimization Framework. In: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research. CASCON '99, IBM Press
17. Beyer, D., Fararooy, A.: A Simple and Effective Measure for Complex Low-Level Dependencies. In: Proceedings of the 8th International Conference on Program Comprehension. ICPC '10, Washington, DC, USA, IEEE Computer Society (2010)
18. Wei, X., Gomez, L., Neamtiu, I., Faloutsos, M.: Permission Evolution in the Android Ecosystem. In: Proceedings of the 28th Annual Computer Security Applications Conference. ACSAC '12, New York, NY, USA, ACM (2012) 31–40
19. Desnos, A., Gueguen, G.: Android: From Reversing to Decompilation. In: Proceedings of the Black Hat Conference, Abu Dhabi, ESIEA: Operational Cryptology and Virology Laboratory (July 2011)
20. Rastogi, V., Chen, Y., Jiang, X.: Droidchameleon: Evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. ASIA CCS '13, New York, NY, USA, ACM (2013) 329–334
21. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA Data Mining Software: An Update. ACM SIGKDD Explorations Newsletter **11**(1) (November 2009) 10–18
22. Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., Hoffmann, J.: Mobile-Sandbox: Having a Deeper Look into Android Applications. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13), New York, NY, USA, ACM (2013) 1808–1815
23. Hosmer, D., Lemeshow, S., Sturdivant, R.: Applied Logistic Regression. 2 edn. Wiley Series in Probability and Statistics. John Wiley & Sons (2013)