Impact of Code Obfuscation on Android Malware Detection based on Static and Dynamic Analysis

Alessandro Bacci[1], Alberto Bartoli[1], Fabio Martinelli[2], Eric Medvet[1], Francesco Mercaldo[2] and Corrado Aaron Visaggio[3]

[1]*Dipartimento di Ingegneria e Architettura, Università degli Studi di Trieste, Trieste, Italy*

[2]*Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Pisa, Italy*

[3]*Dipartimento di Ingegneria, Università degli Studi del Sannio, Benevento, Italy*

Keywords: Malware, Android, Machine Learning, Code Obfuscation, Security.

Abstract: The huge diffusion of malware in mobile platform is plaguing users. New malware proliferates at a very fast pace: as a matter of fact, to evade the signature-based mechanism implemented in current antimalware, the application of trivial obfuscation techniques to existing malware is sufficient. In this paper, we show how the application of several morphing techniques affects the effectiveness of two widespread malware detection approaches based on Machine Learning coupled respectively with static and dynamic analysis. We demonstrate experimentally that dynamic analysis-based detection performs equally well in evaluating obfuscated and non-obfuscated malware. On the other hand, static analysis-based detection is more accurate on non-obfuscated samples but is greatly negatively affected by obfuscation: however, we also show that this effect can be mitigated by using obfuscated samples also in the learning phase.

1 INTRODUCTION

Malware targeting mobile platforms has been spreading fastly and largely in the last years. This is an natural consequence of two facts, which constitute strong incentives for many attackers: (i) users store more and more sensitive and private information in their mobile devices and (ii) mobiles, and Android-bases in particular, are becoming the most used devices: in March 2017, Android usage hit 37.93% while Windows on computers hit 37.91%[1].

This is the reason why antimalware vendors propose free and commercial solutions with the aim to mitigate this widespread phenomenon, but the current signature-based approach is not sufficient to protect users against the new threats developed by malware writer (Canfora et al., 2015b; Rastogi et al., 2013a; Zheng et al., 2013). As a matter of fact, signature-based malware detection (the most common technique adopted by mobile antimalware) is often ineffective (Cimitile et al., 2017). Moreover it is costly: the process for obtaining and classifying a malware signature is laborious and time-consuming.

[1]http://gs.statcounter.com/os-market-share#monthly-201703-201703-map

In the last years, the research community has developed several methods in order to identify whether a mobile application exhibits a malicious behaviour: basically the approaches considered are based on static analysis (the detection process does not require the execution of the application) or on dynamic analysis (the detection process requires the application to run in order to identify the maliciousness) (Tam et al., 2017).

While several research papers evaluate the effectiveness of the signature-based detection provided by current antimalware technologies (Zheng et al., 2013; Ramachandran et al., 2012; Rastogi et al., 2013a,b), in this paper our aim is to evaluate the effectiveness of the techniques considered by researchers against the common code morphing techniques employed by malware writers. In order to demonstrate this, we evaluate two recent approaches based on Machine Learning techniques operating on, respectively, features derived from static analysis (Canfora et al., 2015a) and dynamic analysis (Canfora et al., 2015c) against a set of widespread morphing techniques. The considered approaches are representative of the many Machine Learning-based malware detection systems which have been recently proposed (e.g., Xue et al.

Bacci, A., Bartoli, A., Martinelli, F., Medvet, E., Mercaldo, F. and Visaggio, C.
Impact of Code Obfuscation on Android Malware Detection based on Static and Dynamic Analysis.
DOI: 10.5220/0006642503790385
In *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP 2018)*, pages 379-385
ISBN: 978-989-758-282-0

(2017); Martinelli et al. (2017); Ferrante et al. (2016); Medvet and Mercaldo (2016); Tam et al. (2017); Backes and Nauman (2017); Demontis et al. (2017)).

The paper poses the following research question: *to which degree the widespread obfuscation techniques affect the effectiveness of state-of-the-art detection approaches for malware detection?* We attempt to answer this question by means of a thorough experimental analysis involving a real-world dataset composed by 3500 legitimate and 3500 real-world malware applications and 8 different and Android-specific morphing techniques.

2 RELATED WORK

There is an increasing interest in applying Machine Learning-based techniques to the problem of Android Malware detection: we here briefly survey the most recent ones, and other non ML-based significant ones, which explicitly consider code obfuscation.

A framework able to inject a set of morphing techniques ha been proposed by Rastogi et al. (2013a) with the aim to evaluate the current antimalware technologies against morphed variants of malware. The main outcome of the paper is that all the studied antimalware software are vulnerable to trivial code transformations.

Rastogi et al. (2014) evaluate ten antimalware tools using six original and morphed mobile malware belonging to six different families. The authors conclude that the antimalware are susceptible to common widespread evasion techniques.

Suarez-Tangil et al. (2016a) propose DroidSieve, an Android malware classifier based on static analysis, and identify two high-level classes: (i) resource-centric features which are derived from resources used by the app and (ii) syntactic features which are derived from the code and metadata of mobile applications. The proposed approach consider obfuscation-invariant features and artefacts introduced by obfuscation mechanisms used by mobile malware writers.

Alterdroid (Suarez-Tangil et al., 2016b) is a malware analysis framework consisting in the analysis of the behavioral differences between the original application and a set of automatically generated versions of it, where a number of modifications have been carefully injected (the so-called variants). In addition, Alterdroid performs a dynamic analysis (i.e., every app was executed over a time span equal to 120 seconds) to identify the malware.

O'kane et al. (2016) investigate the optimal set of instruction being executed to identify obfuscated Android malware using the SVM classifier. They find a set of instructions that are good indicators of malware and determine how long the program needs to run in order to obtain an accurate classification. They obtain an average accuracy equal to 84.4%.

The RevealDroid tool (Garcia et al., 2015) is stated to be obfuscation resilient thanks to a set of features including sensitive APIs and intents usage and information flows. The effectiveness of the selected features is evaluated using two different simple classifiers, which obtain an accuracy ranging between 93% and 96% in malware detection.

3 MACHINE LEARNING-BASED MALWARE DETECTION

We consider two forms of detection based on Machine Learning techniques applied on data derived from static and dynamic analysis, i.e., on sequences of opcodes and system calls, respectively. We build our study on two state-of-the-art approaches (Canfora et al., 2015a,c) which we briefly describe in the following sections.

In both cases, the approach consists of a *classification phase*, in which an input application a is classified as malware or trusted, and a *learning phase* in which a classifier is trained basing on two sets A_T and A_M including, respectively, trusted and malware applications. In both phases, a numeric feature vector is computed out of the app a by means of a *pre-processing step*. All the procedures are described below.

3.1 Static Analysis

The pre-processing of an app a starts by extracting the `.dex` file from a packed as an `.apk` file. Then, several files containing the machine level instructions, each consisting in an opcode and its parameters, are obtained from the `.dex` file by means of decompilation. From these files, a list of sequences of opcodes (without the parameters), where a sequence corresponds to a method of a class in the app, is extracted. Finally, the frequency $f(a,o)$ of each $n$gram o occurring in the sequences of the list is computed, n being a parameter of the method. The resulting vector is the initial feature vector for a.

In the learning phase, an feature selection procedure is performed, since the feature vector obtained through the pre-processing phase may be remarkably large. This is done proceeding as follow. For each $n$grams o, its global frequencies relatively to A_T (set of trusted apps) and A_M (set of malware apps) are computed:

$$\bar{f}_M(o) = \frac{1}{|A_M|} \sum_{a \in A_M} f(a,o) \quad (1)$$

$$\bar{f}_T(o) = \frac{1}{|A_T|} \sum_{a \in A_T} f(a,o) \quad (2)$$

The relative difference $d(o)$ is obtained as:

$$d(o) = \frac{\text{abs}(\bar{f}_M(o) - \bar{f}_T(o))}{\max(\bar{f}_M(o), \bar{f}_T(o))} \quad (3)$$

The set O of the selected *n*grams (and hence the corresponding features) is hence built to include the k *n*grams with the highest values of $d(o)$, where k is a parameter of the method. The *n*grams for which $d(o) = 1$ (i.e., those *n*grams which occur only in A_T and not in A_M or viceversa) are not considered to avoid obtaining a classifier that fails to generalize. Furthermore, all the *n*grams in O that are subsequences of another *n*gram in O are discarded—this way, redundant information is removed. Finally, only the $k' < k$ *n*grams in O with the highest $d(o)$ are retained, where k' is a parameter of the method. At the end of the learning phase, a binary classifier based on Support Vector Machine (SVM) with a Gaussian kernel and a cost $c = 1$ is learned on the dataset deriving from A_T, A_M and the features determined by O.

In the classification phase, the feature vector for the input app a is first computed considering the frequences of the opcodes in O; then, it is given as input to the trained SVM which outputs a response in {malware, trusted}.

3.2 Dynamic Analysis

In the pre-processing, the system calls invoked by the app a during the execution are recorded, producing an execution trace. Then the feature vector is extracted calculating the frequency over a of each possible *n*gram of system calls (w/o the call arguments), where n is a parameter of the method.

As in the static case, the learning phase starts with a feature selection procedure. To reduce the number of features, only the k *n*grams with the greatest δ_s are selected, with:

$$\delta_s = \frac{\left| \frac{1}{|A_T|} \sum_{a \in A_T} f(a,s) - \frac{1}{|A_M|} \sum_{a \in A_M} f(a,s) \right|}{\max_{a \in A} f(a,s)}$$

where s is an *n*gram of system calls, A is the union of A_T (set of trusted apps) and A_M (set of malware apps), and k is a parameter of the method. The number of features is further reduced by computing, for each remaining s, the mutual information I_s of $f(a,s)$ with the label of a for any $a \in A$ and retaining the k' features with the highest I_s, with k' being a parameter of the method, resulting in a set $\mathcal{S}$ of selected *n*grams. At the end of the learning phase, a binary classifier based on Support Vector Machine (SVM) with a Gaussian kernel and a cost $c = 1$ is learned on the dataset deriving from A_T, A_M and the features determined by $\mathcal{S}$.

In the classification phase the previously selected features are extracted from the apps in the unlabelled dataset, on which the trained classifier is applied, receiving a response label in {malware, trusted}.

4 EXPERIMENTAL EVALUATION

4.1 Data

We built a dataset of 7000 applications evenly divided between trusted ($\mathcal{A}_T$) and malware ($\mathcal{A}_M$). In particular, we took a subset of the dataset used in (Canfora et al., 2015a) in which trusted apps were collected from Google Play and malware apps from the Drebin dataset Arp et al. (2014). Furthermore, we built a set $\mathcal{A}_O$ of obfuscated malware apps set by applying to each of the apps in $\mathcal{A}_M$ all the obfuscation techniques described in the next section.

For the dynamic analysis detection, we executed each app on a real Android device for at most 1 minute, during which a tool was simulating random UI interactions for the whole minute of execution.

4.2 The Obfuscation Techniques

Android runs *Dalvik* executables stored in `.dex` files. In order to apply transformations to application code, we obtained the *smali* (a human readable dalvik bytecode) representation of the code, using *apktool*[2], a tool for reverse engineering which allows to decompile and recompile Android applications. apktool is able to decode resources to nearly original form and rebuild them after making some modifications. The smali representation is the target of the transformations we considered.

We designed, implemented, and publicly released[3] a Java tool able to apply a set code modifications to smali representation in an automated way.

We applied all the following morphing techniques:

1. **Disassembling & Reassembling.** The compiled Dalvik bytecode in `classes.dex` of the application package may be disassembled and reassem-

[2]http://ibotpeaches.github.io/Apktool/

[3]https://github.com/faber03/AndroidMalwareEvaluatingTools

bled through *apktool*. This allows various items in a `.dex` file to be represented in a different way. In this way signatures relying on the order of different items in the `.dex` file will likely be ineffective with this transformation.

2. **Repacking.** Every Android application contains a developer signature key that will be lost after disassembling the application and then reassembling it. In order to create a new key we consider the *signapk*[4] tool to embed a new signature key in the reassembled app to avoid detection signatures that match the developer keys.
3. **Changing Package Name.** Each Android application is identified by a unique package name. This transformation is focused at renaming the application package name in both the Android Manifest and all the classes of the app, to elude detection by signatures based on package name.
4. **Identifier Renaming.** To avoid detection signatures relying on identifier names, this transformation renames each package name and class name by using a random string generator, in both Android Manifest and smali classes, handling renamed classes invocations.
5. **Data Encoding.** The `dex` files contain all the strings and arrays used in the code. Strings could be used to create detection signatures to identify malware. To elude such signatures, this transformation encodes strings with a *Caesar cipher* with a fixed key equal to 3. This technique is also applied to the code of the so-called metamorphic malware Borello and Mé (2008); Canfora et al. (2014). The original string will be restored during application run-time.
6. **Call Indirections.** Some detection signatures could exploit the call graph of the application. To evade such signatures we designed a transformation which mutates the original call graph, by modifying every method invocation in the smali code with a call to a new method inserted by the transformation which simply invokes the original method.
7. **Code Reordering.** This transformation is aimed at modifying the instructions order in smali methods. A random reordering of instructions has been accomplished by inserting `goto` instructions with the aim of preserving the original runtime execution trace. Considering that the reordering is random, this is considered the strongest obfuscation technique able to alter the signature provided by current antimalware technologies You and Yim (2010). The transformation was applied only to methods that do not contain any type of jumps (i.e., `if`, `switch`, recursive calls).
8. **Junk Code Insertion.** These transformations introduce code sequences that have no effect on the business logic of applications. This is considered a weak technique, for this reason usually antimalware technologies can be able to identify samples obfuscated only with this technique Collberg et al. (2003). The transformation provides three different junk code insertions: (i) insertion of `nop` instructions into each method, (ii) insertion of unconditional jumps into each method, and (iii) allocation of three additional registers on which garbage operations are performed.

[4]https://code.google.com/p/signapk/

4.3 Procedure and Results

We performed a 10-fold cross validation, i.e., we: (i) randomly split the sets $\mathcal{A}_T$ and $\mathcal{A}_M$ in 10 partition; (ii) built the sets A_T and A_M by including 9 on the 10 partitions in $\mathcal{A}_T$ and $\mathcal{A}_M$, respectively; (iii) we performed the learning phase on A_T and A_M, as described in Sections 3.2 and 3.2; (iv) for each $a \in \mathcal{A}_T \cup \mathcal{A}_M \cup \mathcal{A}_O$ and not in $A_T \cup A_M$ (i.e., for each app in the *testing set*), we applied the learned classifier.

We repeated steps ii, iii, and iv 10 times by varying the excluded partition. For the dynamic case, we collected 10 execution traces for each app (used both in the learning and classification phases, with traces for the same app randomly distributed in the learning and testing sets) in order to mitigate the impact of fortunate and unfortunate conditions during the execution. We set $n = 3$, $k = 5000$, and $k' = 2000$ for the static case and $n = 3$, $k = 2000$, and $k' = 750$ for the dynamic case, basing on the results of the two corresponding original papers.

We measured the classification effectiveness in terms of Accuracy, i.e., the percentage of correctly classified apps, False Positive Rate (FPR), i.e., the percentage of trusted apps classified as malware, and False Negative Rate (FNR), i.e., the percentage of malware apps classified as trusted. All the results are shown in Table 1: FNR is cast as $\text{FNR}_{\neg O}$ and FNR_O, i.e., measured on non-obfuscated malware apps ($a \in \mathcal{A}_M \setminus A_M$) and obfuscate malware apps ($a \in \mathcal{A}_O$), respectively. Figure 1 shows True Positive Rate (TPR) and True Negative Rate (TNR) indexes for each of the 10 repetitions—TNR is the average of $\text{TNR}_{\neg O}$ and TNR_O obtained in the repetition.

It can be seen from Table 1 that both methods (i.e., static anlysis-based and dynamic analysis-based detection) are effective in classifying non-obfuscated

Table 1: Mean μ and standard deviation σ of FNR and FPR across the 10 repetitions in the two learning scenarios: without (above) or with (below) obfuscated malware apps in the training set.

		FPR		FNR$_{\neg O}$		FNR$_O$	
	Method	μ	σ	μ	σ	μ	σ
w/o	Static	3.7	1.3	2.6	0.8	89.8	0.2
w/o	Dyn.	9.9	1.0	5.8	1.4	7.5	0.3
w/	Static	6.6	1.8	0.6	0.1	0.1	0.1
w/	Dyn.	10.7	1.0	3.2	0.2	4.5	0.2

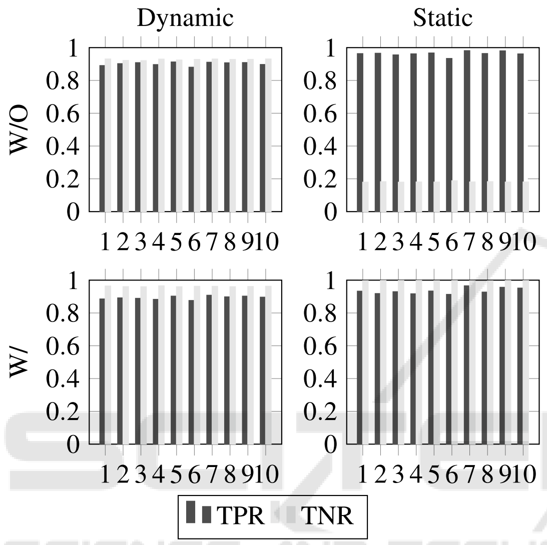

Figure 1: Effectiveness for each of the 10 repetitions in term of TPR and TNR of both static and dynamic analysis. Results are shown in the case with and without obfuscated malware in the training set.

apps, FPR and FNR$_{\neg O}$ being lower than 10%. Static analysis is indeed more accurate, with an FPR $< 4\%$ and FNR$_{\neg O}$ $< 3\%$, whereas dynamic analysis scores $\approx 10\%$ and 6% respectively: the accuracy of the latter is negatively affected by the variability of executions which essentialy results in noisy data. These figures are consistent with the results of Canfora et al. (2015a) and Canfora et al. (2015c).

The most interesting finding concerns, however, the impact of obfuscation on malware detection. By observing the difference between FNR$_{\neg O}$ and FNR$_O$ in the two topmost rows of Table 1, it can be seen that the effectiveness of static analysis-based detection is severely affected by obfuscation, whereas dynamic analysis-based effectiveness is not significantly affected. For static method, FNR$_O$ $\approx 90\%$, i.e., 9 on 10 malware apps are wrongly classified as trusted. This can be explained by the fact that the obfuscation techniques applied in this study heavily modify frequency and order of the opcodes in an app, especially in the case of Call indirections, Code reordering, and Junk code insertion. This leads to a completely different distribution of *n*grams that is no longer recognized by the static classifier. Instead, execution traces of an obfuscated app are very similar to their non-obfuscated counterpart, therefore the dynamic classifier is not influenced by obfuscation. In essence, this experiment confirms the high level intuition that dynamic analysis-based detection is much more robust to code obfuscation than static one.

4.3.1 Learning on Obfuscated Malware

Basing on the results of our first experimentation, we decided to investigate if the scarce robustness to obfuscation of the static analysis-based detection may be mitigated. In other words, we tried to address the high level research question: *are features based on ngrams of obcodes able to capture the essence of malware even in case of obfuscation?* To answer this question experimentally, we modified the experimental procedure such that the learning set A_M consists of an even number of apps from the set $\mathcal{A}_M$ of non-obfuscated malware apps and from the set $\mathcal{A}_O$ of obfuscated malware apps, with $|A_M| = |A_T|$—again, apps used for learning are never used for assessing classification effectiveness.

Table 1 presents—in the two bottom rows—the results in terms of FPR, FNR$_{\neg O}$, and FNR$_O$ of the two methods with the obfuscated malware apps in the learning set.

It can be seen that simply making obfuscated malware available to the learning process makes static analysis-based detection clearly robust to obfuscation: FNR$_{\neg O}$ and FNR$_O$ are both very low (0.6% and 0.1%, respectively), whereas FPR is only slightly higher than with the case of non-obfuscated only learning. In other words, features based on *n*grams of obcodes are adequate for capturing the essence of malware regardless of obfuscation, but samples of obfuscated malware must be available for the learning phase.

Concerning the dynamic method, effectiveness indexes deviate only moderately with, in general, lower FNR and higher FPR.

5 CONCLUSION AND FUTURE WORK

In this work, we compared the robustness to code obfuscation of two different malware detection methods, based on Machine Learning techniques applied on features deriving from static (opcodes in machine leavel app code) and dynamic (system calls in app execution trace) analysis. The underlying assumption is that obfuscating the code of an app should leave its execution trace almost unchanged, making a dynamic classifier robust to obfuscation, but should change completely the sequence of opcodes deriving from its code, making a static classifier totally ineffective. We experimentally validated this assumption by applying two state-of-the-art methods to legitimate apps, malware apps, and malware apps subjected to 8 different code morphing techniques: results show that static analysis-based detection is essentially uneffective on obfuscated malware. We also showed that static detection may be made robust to obfuscation by making obfuscated malware apps available for the learning. In the future, we plan to study if and to which degree static and dynamic detection are able to correctly classify apps subjected to *new* code morphing techniques, i.e., techniques for which no samples were available in the learning phase.

ACKNOWLEDGEMENTS

This work has been partially supported by H2020 EU-funded projects NeCS and C3ISP and EIT-Digital Project HII.

REFERENCES

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., and Siemens, C. (2014). Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*.

Backes, M. and Nauman, M. (2017). Luna: Quantifying and leveraging uncertainty in android malware analysis through bayesian machine learning. In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*, pages 204–217. IEEE.

Borello, J.-M. and Mé, L. (2008). Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220.

Canfora, G., De Lorenzo, A., Medvet, E., Mercaldo, F., and Visaggio, C. A. (2015a). Effectiveness of opcode ngrams for detection of multi family android malware. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 333–340. IEEE.

Canfora, G., Di Sorbo, A., Mercaldo, F., and Visaggio, C. A. (2015b). Obfuscation techniques against signature-based detection: a case study. In *Mobile Systems Technologies Workshop (MST), 2015*, pages 21–26. IEEE.

Canfora, G., Medvet, E., Mercaldo, F., and Visaggio, C. A. (2015c). Detecting android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 13–20. ACM.

Canfora, G., Mercaldo, F., Visaggio, C. A., and Di Notte, P. (2014). Metamorphic malware detection using code metrics. *Information Security Journal: A Global Perspective*, 23(3):57–67.

Cimitile, A., Martinelli, F., Mercaldo, F., Nardone, V., and Santone, A. (2017). Formal methods meet mobile code obfuscation identification of code reordering technique. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2017 IEEE 26th International Conference on*, pages 263–268. IEEE.

Collberg, C. S., Thomborson, C. D., and Low, D. W. K. (2003). Obfuscation techniques for enhancing software security. US Patent 6,668,325.

Demontis, A., Melis, M., Biggio, B., Maiorca, D., Arp, D., Rieck, K., Corona, I., Giacinto, G., and Roli, F. (2017). Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Transactions on Dependable and Secure Computing*.

Ferrante, A., Medvet, E., Mercaldo, F., Milosevic, J., and Visaggio, C. A. (2016). Spotting the malicious moment: Characterizing malware behavior using dynamic features. In *Availability, Reliability and Security (ARES), 2016 11th International Conference on*, pages 372–381. IEEE.

Garcia, J., Hammad, M., Pedrood, B., Bagheri-Khaligh, A., and Malek, S. (2015). Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. *Department of Computer Science, George Mason University, Tech. Rep*.

Martinelli, F., Marulli, F., and Mercaldo, F. (2017). Evaluating convolutional neural network for effective mobile malware detection. *Procedia Computer Science*, 112(C):2372–2381.

Medvet, E. and Mercaldo, F. (2016). Exploring the usage of topic modeling for android malware static analysis. In *Availability, Reliability and Security (ARES), 2016 11th International Conference on*, pages 609–617. IEEE.

O'kane, P., Sezer, S., and McLaughlin, K. (2016). Detecting obfuscated malware using reduced opcode set and optimised runtime trace. *Security Informatics*, 5(1):1–12.

Ramachandran, R., Oh, T., and Stackpole, W. (2012). Android anti-virus analysis. In *Annual Symposium on Information Assurance & Secure Knowledge Management*, pages 35–40.

Rastogi, V., Chen, Y., and Jiang, X. (2013a). Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM.

Rastogi, V., Chen, Y., and Jiang, X. (2013b). Droidchameleon:evaluating android anti-malware against transformation attacks. In *ACM Symposium on Information, Computer and Communications Security*, pages 329–334.

Rastogi, V., Chen, Y., and Jiang, X. (2014). Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108.

Suarez-Tangil, G., Dash, S. K., Ahmadi, M., Kinder, J., Giacinto, G., Cavallaro, L., Dash, S. K., Suarez-Tangil, G., Khan, S., Tam, K., et al. (2016a). Droidsieve: Fast and accurate classification of obfuscated android malware. In *Proc. 5th {ACM} Conf. Data and Application Security*, volume 7148, pages 43–50. IEEE.

Suarez-Tangil, G., Tapiador, J. E., Lombardi, F., and Di Pietro, R. (2016b). Alterdroid: differential fault analysis of obfuscated smartphone malware. *IEEE Transactions on Mobile Computing*, 15(4):789–802.

Tam, K., Feizollah, A., Anuar, N. B., Salleh, R., and Cavallaro, L. (2017). The evolution of android malware and android analysis techniques. *ACM Comput. Surv.*, 49(4):76:1–76:41.

Xue, Y., Meng, G., Liu, Y., Tan, T. H., Chen, H., Sun, J., and Zhang, J. (2017). Auditing anti-malware tools by evolving android malware and dynamic loading technique. *IEEE Transactions on Information Forensics and Security*.

You, I. and Yim, K. (2010). Malware obfuscation techniques: A brief survey. In *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*, pages 297–300. IEEE.

Zheng, M., Lee, P. P. C., and Lui, J. C. S. (2013). Adam: An automatic and extensible platform to stress test android anti-virus systems. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, pages 82–101.