Evasion Is Not Enough: A Case Study of Android Malware

Harel Berger[1](✉), Chen Hajaj[2], and Amit Dvir[3]

[1] Ariel Cyber Innovation Center, CS Department, Ariel University, Kiryat Hamada, 40700 Ariel, Israel
harel.berger@msmail.ariel.ac.il

[2] Ariel Cyber Innovation Center, Data Science and Artificial Intelligence Research Center, IEM Department, Ariel University, Ariel, Israel
chenha@ariel.ac.il
https://www.ariel.ac.il/wp/chen-hajaj/

[3] Ariel Cyber Innovation Center, CS Department, Ariel University, Ariel, Israel
amitdv@ariel.ac.il
https://www.ariel.ac.il/wp/amitd/

Abstract. A growing number of Android malware detection systems are based on Machine Learning (ML) methods. However, ML methods are often vulnerable to *evasion attacks*, in which an adversary manipulates malicious instances so they are classified as benign. Here, we present a novel evaluation scheme for evasion attack generation that exploits the weak spots of known Android malware detection systems. We implement an innovative evasion attack on Drebin [3]. After our novel evasion attack, Drebin's detection rate decreased by 12%. However, when inspecting the functionality and maliciousness of the manipulated instances, the maliciousness rate increased, whereas the functionality rate decreased by 72%. We show that non-functional apps, do not constitute a threat to users and are thus useless from an attacker's point of view. Hence, future evaluations of attacks against Android malware detection systems should also address functionality and maliciousness tests.

Keywords: Cyber security · Android security · Malware detection

1 Introduction

Android malware evolves over time, such that malicious versions of popular Android application PacKages (APKs) can propagate to various Android markets. One of the most popular techniques in the malware detection domain is Machine Learning (ML) based detection of malicious entities, where some techniques' detection rates exceed 99% at times. However, Szegedy et al. [7] showed that some ML methods (including malware detection systems) are vulnerable to *adversarial examples*. A special case of adversarial examples involves using *evasion attacks*. Evasion attacks take place when an adversary modifies malware code so that the modified malware is categorized as benign by the ML, but still successfully executes the malicious payload.

The main contribution of this work is our analysis of functionality and maliciousness tests for Android evasion attacks that change existing features. To the best of our knowledge, none of the evasion attacks that changed existing features of malicious apps has been evaluated in terms of either functionality or malicious content maintenance, which raises the question of whether "efficient" attacks result in nonfunctional apps that would not be malicious as intended.

2 Related Work

In this section, we survey well known ML-based detection systems for Android malware. We also depict several popular evasion attacks targeting the detection systems. Then, we examine the functionality and maliciousness tests of Android malware. As far as we know, they have not been fully explored in previous studies in the Android malware field.

2.1 Android Malware ML-Based Detection Systems

One of the best known Android malware detection systems is Drebin [3], a lightweight Android malware detector (it can be installed on mobile phones). Drebin collects 8 types of features from the APKs. From the Manifest file, Drebin extracts permissions requests, software/hardware components and intents, and from the smali code, it extracts suspicious/restricted API calls, used permissions in the app's run and URL addresses. A different approach is found in MaMaDroid [14], which extracts features from the Control Flow Graph (CFG) of an application. MaMaDroid creates a tree of API calls based on package and family names. After abstracting the calls, the system analyzes the API call sequence performed by an app, to model its true nature. The third approach which inspects system features was introduced in Andromaly [19].

2.2 Evasion Attacks on ML-Based Detection Systems

Evasion attacks against ML-based detection systems can take multiple courses. One course of action is to camouflage specific parts of the app. One well-known example of camouflage is the use of obfuscation or encryption, which was implemented in Demontis et al. [6]. Reflection, which allows a program to change its behavior at runtime, is also a classic evasion method, which was exampled in Rastogi et al. [16]. A typical approach to evasion attacks on ML-based detection systems involves adding noise to the app, thus misleading the classifier's assignment of benign and malicious app. An example of the use of this approach can be found in Android HIV [5] where the authors implemented non-invoked dangerous functions against Drebin and a function injection against MaMaDroid. Changing the app flow is another approach, where a detection system that is based on analyzing the app flow, such as MaMaDroid, fails to detect the malicious app [5,11].

2.3 Functionality and Maliciousness Tests

An evasion attack's main goal is to evade detection by malware detection systems. However, two tests can be conducted to identify an attack. The first is functionality, and the second is maliciousness. These features were explored in Repackman [18], a tool that automates the repackaging attack with an arbitrary malicious payload. The authors performed 2 relevant tests for their repackaging attack, all using Droidutan [17]. For the functionality test (which they termed **feasibility**), they used random UI actions. For maliciousness, which they termed **reliability**, they measured the apps that successfully executed their payload via recorded screenshots. To the best of our knowledge, these functionality and malicious content tests have not been mentioned in previous evasion attack studies.

3 Evaluation Metrics

This study involves a number of metrics. These metrics are used to define our app dataset, evaluate the effectiveness of the attacks against the detection systems, and formulate insights about our findings. First, we describe the dataset we gathered and its verification tool. In addition, we discuss functionality and maliciousness tests for Android evasion attacks. The metrics are:

- **Data collection:**
  - **Benign apps:** We combine apps from the AndroZoo dataset [1], chosen from the GooglePlay [8] market, and Drebin's [3] dataset.
  - **Malicious apps:** We use the malicious apps from the Drebin dataset [3], CICAndMal2017 [12], AMD [22] and StormDroid [4] datasets.
  - **Verification:** We use VirusTotal (VT) [21] to verify that our apps are correctly labeled. We define benign apps as apps that are not marked as malicious by any scanner. Malicious apps are apps that are identified as malicious by at least 2/4 scanners [3,15]. In our study, we use malicious apps that are identified by at least two scanners.
- **Eliminating dataset biases:** Android ML-based detection systems suffer from temporal and spatial biases [15]. Spatial bias refers to unrealistic assumptions about the ratio of benign to malicious apps in the data. Temporal bias refers to temporally inconsistent evaluations that integrate future knowledge about the test data into the training data. To avoid these dataset biases, we follow the properties suggested in [15]. For example, 90% of our dataset is composed of benign APKs, and the remaining 10% is malicious, similar to the distribution of global apps [9,13], thus accounting for the spatial bias. To prevent temporal bias, we train the classifiers with apps whose timestamp is prior to the test data.
- **Robustness evaluation:** To evaluate robustness, we compute the proportion of instances for which the classifier was evaded; this is our metric of *evasion robustness*, with respect to the robustness of the detection system (similar to the analysis provided in [20]). Thus, evasion robustness of 0% means that the classifier is successfully evaded every time, whereas evasion

robustness of 100% means that the evasion fails in every instance. We compare the evasion robustness of our evasion attack compared to the original malicious APKs detection rate.
- **Functionality:** In an evasion attack, the attacker tries to conceal the malicious content from the detection system. Constructing an evasion attack with the use of existing Android malware includes manipulation of the APK, which is a sensitive task. It may result in any kind of error, thus resulting in a crash. If the crash occurs in the initial stages of the app's run, the malicious content will probably not harm the user at all. Therefore, evaluation of the apps' functionality is vital when generating an APK evasion attack. The functionality check includes a test where the app is installed and run on an Android emulator [10]. If the app crashes, this suggests it is not functional. A manipulated app that passes this test is declared a functional app.
- **Malicious Activity/Maliciousness:** While a manipulated app that passes the previous check can be considered a threat to the user, it does not guarantee that the previous malicious content will run similarly to the previous Android malware. Therefore, a test for the maliciousness of the app is needed. We evaluate malicious activity with the following test: We scan the app using VT. If the number of scanners that indicate the manipulated app to be malicious is less or equal to the number of scanners identifying the original app as malicious, this app is said to pass our simple maliciousness test.

4 Case Study

In this section, we demonstrate the use of our tests. We chose Drebin as a test case for this inquiry, which is one of the best known Android malware detection systems. For a full description of the Drebin classifier, see [3] (implementation is available at [2]). First, we depict our attack, which decreases Drebin's accuracy. Then, we run our additional tests (see Sect. 3) on the original and manipulated apps.

4.1 Attacker Model

The attacker has a black-box access to the input and output of the trained classifier. As can be seen in Fig. 1, The first input (denoted 1 in Fig. 1) for the classifier is malicious APK samples available to the attacker. Based on this input, the Drebin classifier outputs a report on these APKs (denoted 2 in Fig. 1), and an accuracy rate of the classification. The attacker receives this report (denoted 3 in Fig. 1), and manipulates the APKs that Drebin identified as malicious, and sends them as a second input to the classifier (denoted 4 in Fig. 1). Note that the attacker aims to modify the APKs such that the final report (denoted 5 in Fig. 1) will label the malicious APKs as benign and the accuracy rate will be lower than the initial accuracy rate. While doing so, it verifies the malicious behavior of the APKs.

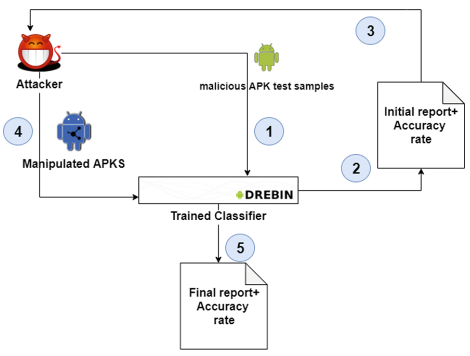
Attacker
malicious APK test samples
Initial report+ Accuracy rate
Manipulated APKS
DREBIN
Trained Classifier
Final report+ Accuracy rate
1
2
3
4
5

Fig. 1. Attacker model. The attacker sends malicious APKs as input to the classifier (1). The classifier produces an initial report and an accuracy rate (2). The attacker receives the output report (3), manipulates the APKs and sends them as a second input to the classifier (4). Finally, the classifier produces a final report and an accuracy rate (5).

4.2 Attack Description

Given malicious APKs to manipulate, and the observations from Drebin, the attacker attempts to conceal the appearance of the observations obtained from the classifier, while still using them to activate the malicious content. It depackages each APK into its subordinate files. We describe the attack in two stages. The input for the first stage is a string. It can resemble a URL address or an API call.

1. **First stage - Encoding:** The attacker analyzes Drebin's observations. It searches for strings in the smali code that are included in the report. The attacker replaces the string with its base64 encoding. It stores the encoded string in the variable of the previous string. Then, the attacker runs a decode function to translate the encoded string back to its previous representation. It stores the decoded string in the previous variable of the string.
2. **Second stage - Reflection:** This stage is only pertinent to encoded strings that resemble an API call, since strings that resemble a URL skip this stage. In this stage, the attacker creates a reflection call. The reflection call creates a general object with the target object from the API call. The next step is invoking the general object with the specific method name from the API call.

5 Results

We evaluated the effectiveness of our attack and the metrics from Sect. 3. We used ∼75 K benign apps and ∼32 K malicious apps (for more details on the source of our data, see Sect. 3). In order to account for a realizable ratio between benign and malicious files [15], we used a proportion of 90/10 between benign and malicious apps in our evaluation. Because we had more malicious apps than one-tenth of the benign apps', we randomly split the malicious data into 5 parts and use the whole benign dataset with each part separately.

In addition, we used a ratio of 80/20 between the train and test data. We did not use the benign samples of the test data while assessing the attack's evasion rate. Overall, we used ∼60 k benign apps and ∼25 k malicious apps as train data, and ∼6 k malicious apps as test data. To account for variations in the data, we used 5-fold CV.

5.1 Evasion Robustness

To get a clearer view of the evasion robustness, we evaluated two cases: the original apps and the evasion attack apps. Any app that triggered an error in the evasion attack's construction was not included in the manipulated test data. Some of the errors we encountered were a result of corrupted APKs that we could not depackage or repackage. Other errors were a consequence of faults in the manipulation process. Overall, the error rate did not exceed 10% of the test data. The results are summarized in Table 1. This table documents the number of malicious apps in each case and the accuracy rate (including the standard deviation). Because the standard deviation was marginal (i.e., <0.02), the evasion robustness rate in each of the splits is similar.

Table 1. Evasion robustness for each case

	Number of applications	Evasion robustness (SD)
Original	6327	0.964 (0.009)
Evasion attack	5817	0.84 (0.012)

5.2 Functionality and Maliciousness Tests

As stated in Sect. 3, our goal was to test whether our evasion attack would damage the functionality and maliciousness of the previous Android malware. We implemented the functionality test on our apps in an emulator using Pixel 2 XL image with SDK 27. We implemented a functionality test on each app before the manipulation (see Sect. 4.2) and after it. For each app, our functionality test was implemented as follows: (1) Cleaning of the emulator log; (2) Installation of the app; (3) Running the app for 1 second; (4) Termination of the app; (5) Uninstallation of the app; (6) Inspection of the emulator log for crashes. We removed ∼28% of the apps that were old or faulty, which led to an interesting insight. Before our evasion attack, 90% of the apps did not crash. After our attack, only 18% of the apps did not crash. Although a nonfunctional app does not attack the user who runs it, we implemented the maliciousness test (see Sect. 3) on the evasion attack apps. After the manipulation, 0.06% of the apps were identified by an equal number of scanners, and 0.1% apps were identified by more scanners than before the manipulation. 99.8% of the apps were identified by a smaller number of scanners than before the manipulation, thus resulting in an increase of ∼9.5 scanners, on average, that did not identify the manipulated apps. While the evasion attack proved to be malicious and decreased the accuracy rate of Drebin, it constitutes a minor threat to the Android user community due to the low level of functionality.

6 Discussion and Conclusion

In this study, we suggested the inclusion of functionality and maliciousness tests in the evaluation of manipulated apps. In addition, we proposed a novel evasion attack that achieved a 12% evasion rate. The maliciousness test we implemented proved that the evasion attack's apps maintained high malicious value, with an additional ∼9.5 VT scanners on average that did not recognize the apps as malicious. However, our functionality test proved that the generated apps' functionality suffered a tremendous loss, from 90% functional apps to 18% functional apps. In a classic analysis, such an attack was considered very powerful and dangerous, and may even result in a urgent update of the classifier. In contrast, the methodology we present proves that the defender has no reason to take steps in the face of such an attack, because its output is deficient in the functional side. To the best of our knowledge, this is the first study to consider these tests for the Android evasion attacks domain. We suggest future works based on our methodology to engineer sophisticated and efficient attacks against well known Android malware detection systems. While doing so, the authors should make sure to maintain high levels of both functionality and maliciousness activity.

Acknowledgement. This work was supported by the Ariel Cyber Innovation Center in conjunction with the Israel National Cyber directorate in the Prime Minister's Office.

References

1. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Androzoo: collecting millions of android apps for the research community. In: Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, pp. 468–471 (2016)
2. Arp, D.: Drebin implementation. github (2014). https://github.com/MLDroid/drebin/
3. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: effective and explainable detection of android malware in your pocket. In: NDSS, vol. 14, pp. 23–26 (2014)
4. Chen, S., Xue, M., Tang, Z., Xu, L., Zhu, H.: Stormdroid: a streaminglized machine learning-based system for detecting android malware. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, pp. 377–388. ACM (2016)
5. Chen, X., et al.: Android HIV: a study of repackaging malware for evading machine-learning detection. IEEE Trans. Inf. Forensics Secur. **15**, 987–1001 (2019)
6. Demontis, A., et al.: Yes, machine learning can be more secure! a case study on android malware detection. IEEE Trans. Dependable Secure Comput. **16**(4), 711–724 (2017)

7. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572 (2014)
8. Google: GooglePlay app market. GooglePlay website (2008). https://play.google.com/store/apps/
9. Google: Android Security 2017 Year In Review. Google website (2017). https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf
10. Google: Run apps on the Android Emulator. Google developers website (2019). https://developer.android.com/studio/run/emulator
11. Ikram, M., Beaume, P., Kaafar, M.A.: Dadidroid: An obfuscation resilient tool for detecting android malware via weighted directed call graph modelling. arXiv preprint arXiv:1905.09136 (2019)
12. Lashkari, A.H., Kadir, A.F.A., Taheri, L., Ghorbani, A.A.: Toward developing a systematic approach to generate benchmark android malware datasets and classification. In: International Conference on Security Technology, pp. 1–7 (2018)
13. Lindorfer, M.: AndRadar: fast discovery of android applications in alternative markets. In: Dietrich, S. (ed.) DIMVA 2014. LNCS, vol. 8550, pp. 51–71. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08509-8_4
14. Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E.D., Ross, G., Stringhini, G.: Mamadroid: detecting android malware by building markov chains of behavioral models. Trans. Privacy Secur. **22**(2), 14 (2019)
15. Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L.: {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In: 28th Security Symposium Security, vol. 19. pp. 729–746 (2019)
16. Rastogi, V., Chen, Y., Jiang, X.: Droid chameleon: evaluating android anti-malware against transformation attacks. In: ACM Symposium on Information, Computer and Communications Security, pp. 329–334. ACM (2013)
17. Salem, A.: Droidutan the android orangutan is a smart monkey that analyzes and tests android applications. https://github.com/aleisalem/Droidutan (2018)
18. Salem, A., Paulus, F.F., Pretschner, A.: Repackman: a tool for automatic repackaging of android apps. In: Proceedings of the 1st International Workshop on Advances in Mobile App Analysis, pp. 25–28. ACM (2018)
19. Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., Weiss, Y.: "andromaly": a behavioral malware detection framework for android devices. J. Intell. Inf. Syst. **38**(1), 161–190 (2012)
20. Tong, L., Li, B., Hajaj, C., Xiao, C., Zhang, N., Vorobeychik, Y.: Improving robustness of {ML} classifiers against realizable evasion attacks using conserved features. In: 28th Security Symposium Security (19), pp. 285–302 (2019)
21. Total, V.: Virustotal-free online virus, malware and url scanner (2012). https://www.virustotal.com/en
22. Wei, F., Li, Y., Roy, S., Ou, X., Zhou, W.: Deep ground truth analysis of current android malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 252–276 (2017)