TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 11/22 pp127–142
DOI: 10.26599/TST.2023.9010005
Volume 29, Number 1, February 2024

Malware Evasion Attacks Against IoT and Other Devices: An Empirical Study

Yan Xu, Deqiang Li, Qianmu Li*, and Shouhuai Xu

Abstract: The Internet of Things (IoT) has grown rapidly due to artificial intelligence driven edge computing. While enabling many new functions, edge computing devices expand the vulnerability surface and have become the target of malware attacks. Moreover, attackers have used advanced techniques to evade defenses by transforming their malware into functionality-preserving variants. We systematically analyze such evasion attacks and conduct a large-scale empirical study in this paper to evaluate their impact on security. More specifically, we focus on two forms of evasion attacks: obfuscation and adversarial attacks. To the best of our knowledge, this paper is the first to investigate and contrast the two families of evasion attacks systematically. We apply 10 obfuscation attacks and 9 adversarial attacks to 2870 malware examples. The obtained findings are as follows. (1) Commercial Off-The-Shelf (COTS) malware detectors are vulnerable to evasion attacks. (2) Adversarial attacks affect COTS malware detectors slightly more effectively than obfuscated malware examples. (3) Code similarity detection approaches can be affected by obfuscated examples and are barely affected by adversarial attacks. (4) These attacks can preserve the functionality of original malware examples.

Key words: Android malware; obfuscation; adversarial examples

1 Introduction

Incorporating edge devices and Artificial Intelligence (AI) has facilitated important advances in the Internet of Things (IoT) domain[1, 2]. The wide employment of IoT devices generates considerable amounts of data[3–6]; thus, traditional cloud computing fails to meet the corresponding demands because of insufficient communication bandwidths and privacy issues. The notion of edge computing can be leveraged to offload tasks from data centers to edge devices, such as cell phones[7, 8] and personal computers[9, 10], to address the aforementioned issues. Correspondingly, AI algorithms have been adapted to edge devices[11], which may only have limited computational power, unstable networking capabilities, and rapidly changing context environments[11–15].

Similar to other kinds of computer systems, edge computing devices are equally vulnerable to various attacks[16–19]. With the increasing storage and use of enterprise data on edge computing devices, these devices will be the major targets of attacks (if not already). One family of cyber threats against edge computing devices is malware. As reported in Ref. [20], the number of malware attacks that target IoT devices reached 56.9 million in 2020, demonstrating an increase of 66% over 2019. Efforts to mitigate the impact of malware, including various Machine Learning (ML) based[21, 22] malware detectors, already exist. However, attackers

• Yan Xu and Qianmu Li are with School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing 210094, China. E-mail: {xuyan, qianmu}@njust.edu.cn.
• Deqiang Li is with School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China. E-mail: lideqiang@njupt.edu.cn.
• Shouhuai Xu is with Department of Computer Science, University of Colorado Colorado Springs, Colorado Springs, CO 80918, USA. E-mail: sxu@uccs.edu.
* To whom correspondence should be addressed.
Manuscript received: 2022-12-01; revised: 2023-01-05; accepted: 2023-01-25

can intentionally transform or manipulate malware examples into some variants, which can evade malware detectors effectively. These attacks are known as evasion attacks[23, 24].

Two major families or classes of evasion attacks are available: obfuscation and adversarial examples. (1) Obfuscation attempts to transform malware examples, preserving functionality but complicating the analysis or understanding of codes[23, 25–30]. There are many readily usable obfuscation tools that can automatically obfuscate software code (e.g., reflection and renaming[25, 26]). Obfuscated malware examples may be effective against detectors that leverage static analysis techniques but not against those that leverage dynamic analysis techniques. (2) Adversarial malware examples have become a popular approach to evading ML-based detectors[24, 31–33]. This approach is often conducted by perturbing a malware example in the feature space and then generating a variant malware example according to the manipulated feature values.

Some studies have analyzed the effectiveness of these attacks[23, 29, 30, 34]. However, there is no systematic study comparing and contrasting the effectiveness of the two classes of attacks. We attempt to fill this void in this paper.

1.1 Our contributions

We empirically evaluate how the two aforementioned classes of attacks may affect IoT and edge computing security. Specifically, we focus on addressing the following questions: (1) How effective are these attacks against Commercial Off-The-Shelf (COTS) malware detectors? (2) How effective are these attacks against code similarity detection approaches? (3) Do these attacks preserve the functionality of the original malware examples?

In order to answer these questions, we use Android malware to conduct our empirical study because Android is a major platform for mobile devices and many applications can be achieved by leveraging Android devices and edge computing devices. Specifically, we consider 10 obfuscation techniques and 9 adversarial malware attacks. The 10 obfuscation techniques are as follows: Class Renaming (CR), Field Renaming (FR), Control Flow (CF), String Encryption (SE), reflection (namely RFL), NOP insertion (namely NOP), Method Renaming (MR), API insertion (namely AINS), Class Insertion (CI), and Permission Insertion (PI). These techniques are scattered in software tools or reported in Refs. [23, 25, 26, 29, 30]. Nine adversarial malware attacks include: Bit Coordinate Ascent (BCA)[35], Bit Gradient Ascent (BGA)[35], Projected Gradient Descent (PGD)[36, 37], Jacobian-based Saliency Map Attack (JSMA)[33, 38], Grosse[32], Gradient Descent with Kernel Density Estimation (GDKDE)[39], Pointwise[40], Salt+Pepper[40], and Mimicry[41].

We analyze the aforementioned obfuscation and adversarial attack techniques considering two Android malware datasets[42]: Drebin[21] and Androzoo[43]. From the Drebin dataset, we randomly select 1108 Android malware examples and use the aforementioned techniques to generate 10 410 obfuscated and 9564 adversarial examples. From the Androzoo dataset, we randomly select 1762 Android malware examples and similarly generate 17 530 obfuscated examples and 15 854 adversarial examples. Our findings are presented as follows. (1) COTS malware detectors offered by VirusTotal can be evaded by evasion attacks. (2) Adversarial malware examples are slightly more effective in evading COTS malware detectors than their obfuscated counterparts. The effectiveness can be attributed to the learning process, which enables adversarial attacks to explore more evasive features than obfuscation attacks. (3) Code similarity detection approaches can be affected by obfuscation attacks but are slightly affected by adversarial attacks. This means that the degree of perturbations incurred by adversarial attacks is smaller than its counterpart incurred by obfuscation techniques. This finding also suggests that using global representation is more reasonable to defend against adversarial attacks than obfuscated examples, which could be leveraged to design future malware detectors. (4) Obfuscated and adversarial malware examples preserve the functionality of original malware to a similar extent.

1.2 Paper outline

The rest of the paper is organized as follows. We review some background knowledge in Section 2. We describe our empirical study design in Section 3. We present our experiments and results in Section 4. Section 5 discusses the results and Section 6 presents the threats to validity. We conclude the paper in Section 7.

2 Background Knowledge

We review Android applications, malware detectors, and evasion attacks (including obfuscated and adversarial attacks) in this section.

2.1 Android applications

An Android app is a set of binary files that can be decompiled (and then recompiled) by using an appropriate reverse engineering tool (e.g., Apktool[†]. An Android app often contains the following.

• **Dalvik bytecodes** are files in the .dex format. These files contain bytecodes implementing the functionalities of the app and can be recognized by an Android virtual machine (e.g., Android Runtime).

• **AndroidManifest. xml** states the necessary information required by an app, including the package name, Android components, permissions, and hardware supports.

• **Resources** are located in the `res` folder and are related to the layout and strings of an app.

• **Assets** are stored in the `assets` folder and contain the files (e.g., web pages) that are not compiled when building an app.

• **Libraries** are stored in the `lib` folder, and comprise complied and platform-dependent libraries.

Attackers can edit an Android app via reverse engineering tools without having access to the source code. For example, Apktool can decode AndroidManifest.xml and disassemble Dalvik bytecodes into smali codes, which is a simplified format for reading disassembled bytecode. The modifications are performed in a certain way such that the modified files can be recompiled to a new APK.

2.2 Android malware detectors

Let $\mathcal{Z}$ denote the set of possible Android apps, $\mathcal{Y} = \{0, 1\}$ denote the label space, where "0" ("1") means an app is benign (malicious). Let $\mathcal{X}$ denote the feature representation space of Android apps, $\phi : \mathcal{Z} \to \mathcal{X}$ denote the feature extractor that maps an app $z \in \mathcal{Z}$ to the representation space $\mathcal{X}$; that is, $\phi(z) = x$ for some $x \in \mathcal{X}$. Let $f : \mathcal{Z} \to \mathcal{Y}$ denote an ML-based malware detector, which takes as input an Android app $z \in \mathcal{Z}$ and outputs a label $y \in \mathcal{Y}$. An ML-based malware detector is a classifier,

$$f = \arg\max_{j \in \mathcal{Y}} F_\theta(\phi(z))_j \tag{1}$$

where $F_\theta : \mathcal{X} \to \mathbf{R}^{|\mathcal{Y}|}$ has a learnable parameter set θ and outputs a confidence score $F_\theta(x)_j$ corresponding to the label $j \in \mathcal{Y}$.

2.3 Modeling obfuscation and adversarial example attacks

Let $\mathcal{M}$ be the set of possible transformations in the feature space that can be applied to a malicious app without changing its functionality. Given a malware example z and a target malware detector f, an evasion attack attempts to manipulate z into z'; therefore f will predict z' as benign. That is,

$$z' \leftarrow z \oplus \delta,$$
$$\text{s.t.,}\ (f(z') = 0) \wedge (f(z) = 1) \wedge (\delta \subseteq \mathcal{M}) \tag{2}$$

where $\oplus$ denotes the operator of applying perturbation δ to z.

We consider two classes of evasion attacks: obfuscation attacks versus adversarial attacks. An obfuscation attack transforms an app into a functionally equivalent one. For example, many obfuscation tools (or obfuscators) are available, which may conduct class renaming, string encryption, and/or debug information removal[23, 25, 26]. Let $\mathcal{M}^o \subseteq \mathcal{M}$ denote the set of possible obfuscation transformations. An obfuscation attack can then be denoted as

$$z' \leftarrow z \oplus \delta^o \tag{3}$$

where $\exists\ \delta^o \subset \mathcal{M}^o$.

By contrast, an adversarial attack transforms an app into a functionally equivalent variant, to evade certain specified malware detectors (e.g., f). In particular, the attacker learns a surrogate model $\hat{f}$ of detector f[41] to facilitate the generation of an adversarial example, where $\hat{f} = f$ if the attacker has access to f. Let $\mathcal{M}^a \subseteq \mathcal{M}$ denote the set of possible adversarial example transformations. An adversarial example attack can be described as follows:

$$z' \leftarrow z \oplus \delta^a,$$
$$\text{s.t.,}\ (\hat{f}(z') = 0) \wedge (\hat{f}(z) = 1) \wedge (\delta^a \subseteq \mathcal{M}^a) \tag{4}$$

Comparison results of Formulas (3) and (4) reveal that obfuscation and adversarial attacks are different in two aspects: (1) they use different transformation sets, namely $\mathcal{M}^o$ vs. $\mathcal{M}^a$; (2) they utilize different strategies to select specific operations and contents for modifications. Considering the transformation set, Table 1 summarizes the transformation sets $\mathcal{M}^o$ and $\mathcal{M}^a$ which have been proposed in Refs. [23, 25, 26, 29, 30, 32, 33, 36, 41, 44–47]. Results show that 9 transformations apply to both attacks, but the other 10 transformations only apply to one of the two attacks.

Specifically, $\mathcal{M}^o$ is often performed on AndroidManifest. xml (e.g., PI), classes. dex (e.g., SE) and other files (e.g., NCE), but $\mathcal{M}^a$ is only conducted on AndroidManifest. xml and/or classes. dex. Moreover, some transformations (e.g., RE and NOP) are popular

† https://ibotpeaches.github.io/Apktool/.

Table 1 Summary of the transformations that have been used by obfuscation attacks and/or adversarial attacks reported in Refs. [23, 25, 26, 29, 30, 32, 33, 36, 41, 44–47]. "Abbr." stands for "Abbrevation". One transformation can manipulate AndroidManifest. xml (dubbed "X" for short), classes.dex (dubbed "D" for short), or other files (dubbed "O" for short) by using either an insertion operation (dubbed "I" for short), removal operation ("R"), or both. We use "✓" ("×") to indicate that a specific transformation is (not) applicable to accomplishing an "I" or "R" operation of a certain file. We list the references of obfuscation/adversarial attacks that apply the specific transformation. We use "○" to indicate that the specific transformation is not implemented by an obfuscation/adversarial attack.

Transformation	Abbr.	Description	Operation		File			Obfuscation attack	Adversarial attack
			I	R	X	D	O		
Manifest randomization	ManR	Randomize entries in AndroidManifest. xml	✓	✓	✓	×	×	Ref. [26]	○
Hardware insertion	HI	Add unused hardware	✓	×	✓	×	×	○	Ref. [32]
Permission insertion	PI	Add unused permission	✓	×	✓	×	×	Ref. [25]	Refs. [41, 47]
Component insertion	CompI	Declare new component	✓	×	✓	×	×	Ref. [25]	Ref. [32]
Intent insertion	IntI	Declare new intent	✓	×	✓	×	×	Ref. [23]	Refs. [32, 47]
Class insertion	CI	Insert dead class	✓	×	✓	×	×	Ref. [25]	Ref. [33]
API insertion	AINS	Insert API	✓	×	×	✓	×	Ref. [25]	Refs. [33, 47]
String insertion	SI	Insert string	✓	×	×	✓	×	Ref. [30]	Refs. [33, 41]
NOP insertion	NOP	Insert "nop" instruction	✓	×	×	✓	×	Refs. [26, 29, 44]	○
Control flow	CF	Insert conditional/loop instruction	✓	×	×	✓	×	Refs. [29, 46]	○
Members reordering	MBR	Reorder variables/method	✓	✓	×	✓	×	Refs. [26, 44]	○
Reflection	RFL	Replace invoke-type instruction by reflection	✓	✓	×	✓	×	Refs. [29, 30]	Ref. [36]
Field renaming	FR	Rename field	✓	✓	×	✓	×	Ref. [44]	○
Method renaming	MR	Rename method	✓	✓	×	✓	×	Ref. [26]	○
Class renaming	CR	Rename package and/or class	✓	✓	✓	✓	×	Refs. [26, 29]	Ref. [36]
String encryption	SE	Encrypt string	✓	✓	×	✓	×	Ref. [45]	Refs. [36, 47]
Class encryption	CE	Encrypt class	✓	✓	✓	✓	×	Refs. [30, 44]	○
Resource encryption	RE	Encrypt resource	✓	✓	×	✓	✓	Ref. [26]	○
Native code encryption	NCE	Encrypt native code	✓	✓	×	✓	✓	Ref. [26]	○

for obfuscation but inapplicable to adversarial attacks. In particular, $\mathcal{M}^a$ is often, if not always, specified in the feature space. Considering attack strategy, obfuscation and adversarial attacks are rule- and learning-based, respectively. Specifically, obfuscation attacks rely on expert-written rules for evasion. For example, some obfuscation tools[25] apply API insertions because API-based detection is a common detection technique. By contrast, adversarial attacks usually learn perturbations from a large set of features by minimizing the confidence score corresponding to the malicious label of the target model f (or the surrogate model $\hat{f}$). For instance, experiments in Ref. [33] show that the insertion of restricted API calls highly contributes to evading the Drebin malware detector[21].

3 Empirical Study Design

This paper aims to investigate the comparative values of obfuscation and adversarial example attacks empirically. This study is decomposed into the following Research Questions (RQs).

- **RQ1: How effective are obfuscation and adversarial example attacks against COTS malware detectors?** Understanding the effectiveness of the state-of-the-art defenses is important in practice. The detection of malware is complicated, and users usually depend on COTS malware detectors to protect their devices. Previous studies have made such an evaluation in the context of obfuscation[23, 29]. To the best of our knowledge, a large-scale evaluation of adversarial attacks is unavailable.

- **RQ2: How effective are obfuscation and adversarial example attacks against the effectiveness of code similarity detection algorithms?** Code similarity is a common approach to detecting repackaged apps[34]. This approach can also be used to detect malware examples as well (e.g., by computing the similarities with known malware examples). Recent studies have investigated the impact of obfuscation attacks[34, 48, 49], but the evaluation of adversarial attacks remains to be systematically investigated.

- **RQ3: Do obfuscated and adversarial examples preserve the functionalities of the original malware examples?** Addressing this question will help understand the side effects of these attacks because a false obfuscation or adversarial perturbation may undermine the invasive capability of resulting malware variants (if executable at all).

We consider obfuscation and adversarial example attacks in the literature to address these RQs. We empirically select representative obfuscators and adversarial attacks, and then produce malware variants, respectively. Evaluation and comparison are performed for each RQ. Figure 1 presents an overview of the study design.

3.1 Selecting obfuscation attacks

We select obfuscation attacks according to the following three criteria: (1) the attack is representative and extensively studied; (2) the attack is open source; (3) the attack can be executed by running a script automatically. Therefore, we consider 10 obfuscation attacks of 2 obfuscation tools: Obfuscapk[26] and AVPASS[25], as shown in Table 2. Obfuscapk and AVPASS decompile the app, modify decompiled files, and then build a new app. We introduce the 10 obfuscation attacks below.

(1) RFL[30]: This attack replace explicit invocations of suitable methods (e.g., public methods but not constructors) with implicit ones by using reflective APIs[26].

(2) SE[30]: This attack replaces a plain string with its encrypted version along with the decryption code snippet (e.g., Obfuscapk[26] uses the AES encryption algorithm for this purpose).

(3) NOP[29]: This attack inserts `nop` instructions randomly into the dexcode.

(4) CR[23]: This attack replaces the name of a class and/or package with some random string and modifies its references correspondingly.

(5) MR[23]: This attack replaces a method's name with some random string and modifies its references correspondingly.

(6) FR[28, 50]: This attack replaces the names of fields and changes the corresponding references (e.g., Obfuscapk[26] renames a field with its MD5 hash value while randomly adding 1–4 fields, and then updates the references correspondingly).

(7) CF[28]: This attack inserts conditional/loop instructions into dex codes (e.g., Obfuscapk inserts `goto` instructions to each method).

(8) AINS[51]: This attack inserts APIs (e.g., AVPASS[25] adds the Android API `DateFormat;-> <init>` into the dexcode code).

(9) PI[23, 51]: This attack inserts permissions into the manifest file (e.g., AVPASS[25] specifies a list of permissions, and randomly selects 15 of them for insertion into AndroidManifest. xml).

(10) CI[29]: This attack inserts some predefined classes into the dexcode (e.g., AVPASS[25] encapsulates several class files and inserts them into the app).

3.2 Selecting adversarial example attacks

An adversarial attack usually contains the following three steps: (1) training an ML-based model, (2) perturbing features in the feature space by using an adversarial algorithm, and (3) modifying apps in the problem space accordingly. We use the following criteria to select adversarial example attacks: (1) an attack outputs Android apps; (2) an attack is effective in evading ML-based malware detectors, as reported in the literature. Therefore, we consider 9 adversarial example attacks against the Drebin detection model.

First, the Drebin detection model is selected because the feature sets contain multiple features, which allows for using multiple transformations. Table 3 presents the Drebin feature sets, along with allowed perturbations

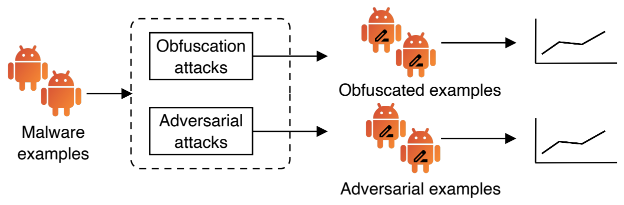


Fig. 1 Overall methodology for evaluating obfuscation and adversarial attacks.

Table 2 Selected obfuscation attacks.

	RFL	SE	NOP	CR	MR	FR	CF	AINS	PI	CI
AVPASS	–	–	–	–	–	–	–	✓	✓	✓
Obfuscapk	✓	✓	✓	✓	✓	✓	✓	–	–	–

Table 3 Overview of Drebin feature sets, along with the manipulations that are used in this paper, where "0 → 1" means feature insertion, "1 → 0" means feature removal, and "✓" ("×") indicates that flipping is (not) allowed.

Feature set			$0 \rightarrow 1$	$1 \rightarrow 0$
Manifest	S_1	Hardware feature	✓	×
	S_2	Requested permission	✓	×
	S_3	App component	✓	✓
	S_4	Filtered intent	✓	×
Dexcode	S_5	Restricted API call	✓	✓
	S_6	Used permission	×	×
	S_7	Suspicious API call	✓	✓
	S_8	Network addresse	✓	✓

in the feature space. The surrogate model is evaluated by two standard metrics. (1) Accuracy (Acc) is the percentage of examples that are classified correctly. (2) F1-score is the weighted average of "precision" and "recall", where "precision" is the percentage of true-positives among the true-positives and false-positives, and "recall" is the percentage of the true-positives among the true-positives and false-negatives.

Second, Nine adversarial example attacks that perturb examples in the feature space are as follows.

(1) BCA[35]: This attack flips a feature value from 0 to 1 if the partial derivative of the loss function with respect to the input is not smaller than a predetermined threshold value, which is the ℓ_2 norm of the derivatives divided by $\sqrt{m}$, where m is the number of dimensions of the input vector. This process is repeated until the predetermined perturbation limit is reached or an adversarial example is successfully generated.

(2) BGA[35]: This attack flips a feature value from 0 to 1 if it corresponds to the max value of the partial derivative of the loss function with respect to the input. This process is repeated until the predetermined perturbation limit is reached or a successful adversarial example is identified.

(3) PGD[36, 37]: This attack searches for perturbations via

$$\delta = \text{Proj}(\delta + \alpha \cdot \nabla_\delta l(f(x + \delta^i), y)) \tag{5}$$

where δ is the perturbation and Proj is the projection onto the ball interest[37]. This process is repeated until the predetermined perturbation limit is reached or a successful adversarial example is identified.

(4) JSMA[33, 38]: This attack flips a feature value from 0 to 1 if this feature is the most important to the prediction of a certain class (e.g., a malicious class for malware detection). This process is repeated until the predetermined perturbation limit is reached or a successful adversarial example is identified.

(5) Grosse[32]: This attack computes the gradient of $F_\theta(x)$ with respect to input x and then flips a feature value from 0 to 1 if the maximal positive gradient is realized. This process is repeated until the predetermined perturbation limit is reached or a successful adversarial example is identified.

(6) GDKDE[39]: This attack produces an adversarial example by solving the following optimization problem:

$$\arg\max_x g(x) = F_\theta(x) - \frac{\lambda}{n} \sum_{i|f(x_i)=0} k\left(\frac{x - x_i}{h}\right) \tag{6}$$

where h is the bandwidth parameter of the Kernel Density Estimation (KDE), k is a kernel function, and n is the number of benign examples.

(7) Mimicry[41]: This attack manipulates a malware example to make it resemble a benign app as much as possible.

(8) Salt+Pepper[40]: This attack perturbs feature vectors with some salt and pepper noises[52].

(9) Pointwise[40]: This attack attempts to reduce the degree of manipulations to a perturbed example while evading the detector.

Third, each perturbation can be implemented by transformations in the problem space. Corresponding elements (i.e., hardware request, permission request, components, and intents) are inserted in AndroidManifest.xml to flip the value of $S_1 - S_4$ from 0 to 1. An app component is removed (i.e., S_3: $1 \rightarrow 0$) by renaming the class name and modifying invocations accordingly. A restricted/suspicious API call is inserted (i.e., S_5 or S_7: $0 \rightarrow 1$) by incorporating the invocation in a dead code (e.g., after the "return" statement).A restricted/suspicious API call is removed (i.e., S_5 or S_7: $1 \rightarrow 0$) by replacing the direct invocation to a reflective one. A network address is inserted in the code, which will be used to flip the value of S_8 from 0 to 1; we encrypt a network address to flip the value of S_8 from 1 to 0.

3.3 Evaluating produced variants

We evaluate obfuscation and adversarial attacks using their impacts on three aspects: malware detection (RQ1), code similarity detection (RQ2), and functionality preservation (RQ3).

In RQ1, we aim to evaluate the effectiveness of obfuscation and adversarial attacks to evade malware detection. The ultimate goal of evasion attacks is to thwart malware detection. Thus, whether an attack can deceive a malware detector is the most concerning, motivating the measurement and comparison of the effectiveness of these attacks. We propose using Evasion Rate (ER), which is the percentage of variants that are misclassified by a detector, to measure the effectiveness. There are two criteria for the selection of target detectors: (1) the target detector is black-box in order that compares these attacks fairly; (2) the target detector can detect Android malware examples effectively in the absence of attacks; otherwise, the evaluation will be meaningless. We consider COTS malware detectors with unknown defense strategies based on criterion (1). We refer to the VirusTotal service‡ because it offers possibly the most comprehensive list of COTS malware detectors. We submit original malware examples to VirusTotal and compute the detection accuracy (i.e., the percentage of detected malware) to achieve criterion (2). The detectors with high accuracy values are selected. Section 4.3 presents additional details.

In RQ2, we aim to evaluate the effectiveness of these attacks to affect code similarity detection approaches. Code similarity detection is the process of marking two apps as a repackaged pair; it is also important for mitigating security threats[34, 50].

In the literature, researchers usually consider the obfuscation resilience of their approaches. To the best of our knowledge, none of the aformentioned literature has considered adversarial attacks. Therefore, in comparison to the impact of obfuscation attacks, we evaluate the impact of adversarial attacks on existing code similarity detection approaches. Specifically, we focus on similarity-based approaches, which are the most common methodology[34]. We compare a malware with its obfuscated/adversarial versions and then observe the average similarity scores to measure their impact. A large similarity indicates a slight impact of the attack; a small similarity indicates a considerable impact[§]. We use Mann–Whitney–Wilcoxon (MWW) to compare the effectiveness of obfuscation and adversarial attacks and investigate if the distributions of their respective similarity scores are statistically significantly different. We also use Cliff's Delta to measure their differences.

We then consider two criteria to select similarity approaches: (1) it is performed statically because we aim to compute similarity scores at a large scale; (2) only AndroidManifest.xml and the codes are considered because our attacks only manipulate these files. Therefore, we select three code similarity detection approaches: Androsim[48], SimiDroid[53], and DroidSim[49]. These approaches compute the similarity between two apps, and mark them as a repackaged pair if the similarity is larger than a threshold (e.g., 0.8 in Ref. [34]).

The Androsim approach[48] measures method-level similarity and scores as follows. First, three kinds of relationships between the two methods are defined: (1) "identical" means that they have the same SHA256 value; (2) "similar" means that their Normalized Compression Distance (NCD)[54] is smaller than or equal to a threshold; (3) "deleted" means that their NCD is higher than the threshold mentioned above. Second, let (a_m, b_m) denote any two similar methods of apps A and B. Let $d_{NCD}(a_m, b_m)$ denote the NCD between a_m and b_m. The similarity score is

$$\text{Androsim} = \frac{\#\text{identical} + \sum\limits_{\forall (a_m, b_m)} (1 - d_{NCD}(a_m, b_m))}{\#\text{identical} + \#\text{similar} + \#\text{deleted}} \tag{7}$$

where #identical, #similar, and #deleted denote the number of identical, similar, and deleted methods, respectively.

The DroidSim approach[49] measures component-wise similarity based on the Component-Based Control Flow Graph (CB-CFG), where nodes and edges represent Android APIs and the control flow precedence of APIs, respectively. Specifically, let $|A_c|$ and $|B_c|$ respectively denote the number of CB-CFGs of apps A and B. For each CB-CFG b_c of app B, let $|b_c| = 1$ if it is isomorphic to any CB-CFG of A, and $|b_c| = 0$ otherwise. The similarity score is as follows:

$$\text{DroidSim}\,(A, B) = \frac{\sum\limits_{\forall b_c} |b_c|}{\min(|A_c|, |B_c|)} \tag{8}$$

SimiDroid[53] computes the similarity between the methods of the two apps as follows. A method is represented by a key/value mapping, where a key is the name of a method and the corresponding value is the combination of the types of statements (e.g., "if" statement) and constants in the method. Let map_1 and map_2 denote the key/value mapping of apps A and B, respectively. The similarity between the two methods is as follows: (1) "identical" if $map_1 = map_2$; (2) "similar" if map_1 and map_2 have the same key but not different values; (3) "different" if $map_1 \neq map_2$. The three kinds of similarities lead to the following: the number of methods that are used in B but not in A is denoted by #new. Meanwhile, the number of methods that are used in A but not in B is denoted by #deleted. Then,

$$\text{SimiDroid} = \max\left(\frac{\#\text{identical}}{\#\text{total} - \#\text{new}}, \frac{\#\text{identical}}{\#\text{total} - \#\text{deleted}}\right) \tag{9}$$

where $\#\text{total} = \#\text{identical} + \#\text{similar} + \#\text{new} + \#\text{deleted}$.

In RQ3, we evaluate to what extent these attacks preserve the functionality of original malware examples.

‡ https://www.virustotal.com/.

§ We do not consider a threshold and metrics like recall in Ref. [34], because there is not a commonly accepted threshold. Instead, it is more straightforward to observe similarity values.

We first justify whether the functionality of a modified malware example is preserved. We consider installability, runnability, and semantics-preservation. (1) A malware example is installable if it can be installed on an Android device. (2) A malware example is runnable if it can be executed without crashing. (3) A malware example preserves the semantics of the original malware if it shows identical behaviors as its original version. We consider display activities and exceptions to measure behaviors. The former is important because display activities are directly observed by a user, and any difference in such activities may draw the attention of a user. The latter is important because many apps throw and log exceptions to identify errors; thus, preserving exceptions would not alert a user.

4 Experiment and Result

4.1 Datasets

Our empirical study is based on two datasets: the Drebin dataset[21] and the Androzoo dataset[43].

Drebin. This dataset contains 5560 malware examples and 123 453 (SHA256 checksums of) benign apps, which were collected between August 2010 and October 2012. This dataset is old but has been widely used in adversarial malware detection studies[32, 33, 36, 41]. These apps are fed to the VirusTotal service to improve their ground truth quality. If at least five scanners label an app as malicious, then we treat it as malware; if no scanner labels an app as malicious, then we treat it as benign (as in previous studies such as Ref. [41]); otherwise, we disregard the app in question. This condition leads to 5560 malware examples (i.e., all the malicious examples in the dataset are malicious) and 37 587 benign apps while disregarding the rest of the apps.

Androzoo. The Drebin dataset is old; therefore, we build an Androzoo dataset based on Refs. [22, 24] as follows. (1) We collect Android examples dated between July 2019 to Dec 2019 from the Androzoo repository[43]. (2) We then determine the ground truth of these examples in the same fashion as in the case of the Drebin dataset. (3) We control the malware ratio to be 10% each month, which is a common practice in Refs. [22, 24]. This condition leads to 9514 malware and 85 658 benign apps.

4.2 Producing original, obfuscated, and adversarial examples

We randomly split each dataset into training, validation, and test sets, demonstrating a 6:2:2 ratio. Figure 2 shows the use of the test set to generate original, obfuscated, and adversarial examples. Specifically, we select the malware examples from each test set that can be successfully repackaged and parsed by Androguard[48]. We refer to the resulting malware examples as the original malware examples to avoid failure of attacks due to the example itself. On the Drebin dataset, 4 examples are filtered from malware examples in the test set, resulting in 1108 original malware examples; on the Andorzoo dataset, 141 examples are filtered from malware examples in the test set, leading to 1762 original malware examples. We then utilize these original malware examples to generate obfuscated malware examples by using the 10 obfuscation attacks (Table 2). Similarly, we use these original malware examples to generate adversarial malware examples by applying the 9 adversarial attacks (Section 3.2). The training and validation sets are used to train the surrogate model of adversarial attacks. The features are normalized by min-max normalization[55]. On the Drebin dataset, the Acc and F1-score values are 99.18% and 96.64%, respectively. On the Androzoo dataset, the Acc and F1 values are 99.19% and 95.86%, respectively.

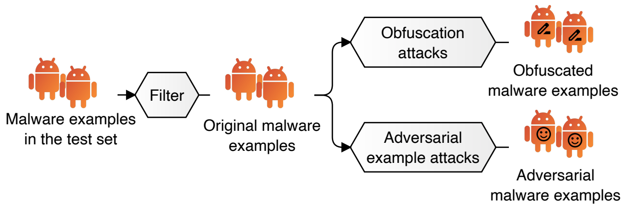

Malware examples in the test set
Filter
Original malware examples
Obfuscation attacks
Adversarial example attacks
Obfuscated malware examples
Adversarial malware examples

Fig. 2 Generation of original, obfuscated, and adversarial malware examples.

Table 4 summarizes the number of original, obfuscated, and adversarial malware examples. From the Drebin test set, we obtain 1108 original malware examples and generate 10 410 obfuscated malware examples and 9564 adversarial malware examples. From the Androzoo test set, we get 1762 original malware examples and generate 17 530 obfuscated malware examples and 15 854 adversarial malware examples.

We observe that these attacks may fail in producing malware variants. We randomly inspect some failures of these attacks and some observations are briefly introduced herein. The AINS obfuscation attack fails when signing the repackaged apps using "jarsigner", and the message is "unable to open jar file". The CR obfuscation attack encounters errors when compiling the repackaged apps because the attack has modified the names of classes but disregarded the associated files in

Table 4 Number of original, obfuscated, and adversarial malware examples, where "–" means no attack is applied.

Example	Attack	Drebin	Androzoo
Original malware	–	1108	1762
Obfuscated malware	AINS	949	1727
	CI	1108	1762
	PI	1108	1728
	CR	986	1757
	MR	915	1761
	FR	919	1758
	SE	1108	1751
	RFL	1105	1762
	NOP	1104	1762
	CF	1108	1762
Adversarial malware	BCA	1108	1762
	BGA	700	1758
	JSMA	1108	1762
	Grosse	1108	1762
	PGD	1108	1762
	GDKDE	1108	1762
	Mimicry	1108	1762
	Pointwise	1108	1762
	Salt+Pepper	1108	1762

the "res" folder.

4.3 Answering RQ1

We evaluate evasion rates of obfuscated and adversarial examples against 20 COTS malware detectors that are offered by the VirusTotal service, as shown in Table 5, to answer RQ1. We select these detectors because they outperform the others in detecting original malware examples (i.e., they achieve high Acc values). Notably, we ignore 9 COTS malware detectors when evaluating the ERs of Androzoo malware variants because the accuracy of these detectors on Androzoo original malware examples is lower than 50% (see Table A1 in Appendix). Evaluating ERs on these detectors would be meaningless.

We submit the obfuscated and adversarial malware examples to VirusTotal for each dataset. We then collect their classification results. Table 6 presents the ERs of

Table 5 Twenty COTS malware detectors from VirusTotal service considered to answer RQ1. The full name of "SymantecMobile" in VirusTotal is "SymantecMobileInsight"; the full name of "McAfee-GW" is "McAfee-GW-Edition".

AVG	Ikarus	ESET-NOD32	Fortinet
Jiangmin	K7GW	SymantecMobile	MaxSecure
Trustlook	Zillya	CAT-QuickHeal	Antiy-AVL
Cyren	AhnLab-V3	McAfee-GW	Symantec
McAfee	MAX	BitDefenderFalx	Cynet

Table 6 Evasion rates of obfuscated (Obf) examples and adversarial (Adv) examples against each COTS malware detector. The full name of "SymantecMobile" in VirusTotal is "SymantecMobileInsight"; the full name of "McAfee-GW" is "McAfee-GW-Edition"; "–" denotes that the evasion rates are ignored because these detectors cannot effectively detect the Androzoo original malware and the evasion rates are meaningless.

(%)

Detector	Drebin dataset		Androzoo dataset	
	Obf set	Adv set	Obf set	Adv set
AVG	4.54	2.75	0	0
Ikarus	0.20	0	0.09	0.01
Fortinet	6.51	1.03	1.05	0.30
ESET-NOD32	2.45	0.74	0.22	0.19
Jiangmin	79.02	84.07	0.10	0.12
K7GW	12.43	0.00	8.32	0.04
SymantecMobile	9.41	9.42	99.12	97.36
MaxSecure	30.58	27.86	23.94	15.57
Trustlook	0.50	0.02	22.52	0.15
Zillya	65.93	81.61	98.27	94.58
Antiy-AVL	62.15	72.15	93.86	93.39
CAT-QuickHeal	3.66	2.76	–	–
Cyren	15.18	13.26	–	–
BitDefenderFalx	0.03	0.17	–	–
Symantec	65.73	74.11	–	–
McAfee-GW	68.66	89.79	–	–
McAfee	61.68	65.30	–	–
MAX	80.96	90.88	–	–
AhnLab-V3	0.88	0	–	–
Cynet	1.08	2.24	–	–
Average	**28.58**	**30.91**	**31.59**	**27.43**

obfuscated and adversarial examples.

First, some COTS malware detectors are heavily evaded when obfuscation or adversarial attacks are applied despite their capability to detect original malware effectively. For example, on the Drebin dataset, MAX can detect original malware with 99.10% accuracy. Meanwhile, MAX is evaded by obfuscated examples with ER = 80.96% and is evaded by adversarial examples with ER = 90.88%. On average, the ERs of obfuscated and adversarial examples are 28.58% and 30.91%, and 31.59% and 27.43% on the Drebin and Androzoo datasets, respectively.

Finding 1: Obfuscation and adversarial attacks can evade COTS malware detectors, with a 30% evasion rate on average.

Second, we observe significant differences of evasion rates among COTS malware detectors. For example, considering the Drebin dataset, ERs of obfuscation and adversarial attacks against Trustlook are 0.50% and 0.02%, respectively; ERs of obfuscation and adversarial attacks against Antiy-AVL are 62.15% and 72.15%, respectively. These discrepancies can be attributed to the different defense strategies used by these COTS malware detectors. The results indicate that some COTS malware detectors (e.g., Trustlook and Ikarus) adopt effective defense strategies against evasion attacks while others (e.g., Antiy-AVL and Zillya) do not. Moreover, some detectors (e.g., MAX and Antiy-AVL) are vulnerable to both kinds of attacks, while some detectors (e.g., Ikarus and K7GW) are robust against both kinds of attacks.

Finding 2: Obfuscation and adversarial attacks have a similar effect on COTS malware detectors.

In addition, adversarial examples achieve a slightly higher evasion rate than obfuscated malware examples. For instance, adversarial examples of the Drebin dataset evade Antiy-AVL with an ER of 72.15%, while obfuscation malware examples achieve 62.15%. This finding may be attributed to the evasive perturbations learned by the adversarial attacks, which are not considered by obfuscation attacks. For instance, the ER of PI obfuscation against Antiy-AVL is 8.49%, and that of BCA is 76.62%. The main difference is that BCA adversarial attack inserts `android.permission.LOCATION` while PI obfuscation attack does not. An obfuscator reasonably disregards inserting this permission because a human may be careful with the permission request. By contrast, BCA finds that the permission helps evade the surrogate model, which also works on Antiy-AVL.

Finding 3: Adversarial malware examples evade COTS malware detectors slightly more often than their obfuscated malware examples.

We observe that adversarial attacks trigger suspicions similarly to obfuscation attacks. For example, AhnLab-V3 fails to detect 14 original malware examples in the Drebin dataset, but can detect their variants produced by all the 9 adversarial attacks and the 3 obfuscation attacks (i.e., NOP, SE, and RFL).

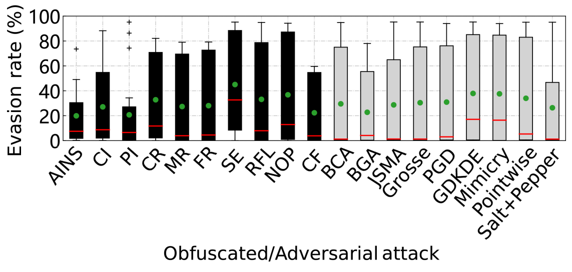
Finding 4: Obfuscation and adversarial attacks can trigger suspicions of COTS malware detectors.

We observe temporal differences in the evasion rates. For example, SymantecMobileInsight is barely evaded by variants of malware examples in the Drebin dataset (ER $<$ 9.5%) but is largely evaded by the malware examples in the Androzoo dataset (ER $>$ 97%). On the contrary, Jiangmin is vulnerable to malware examples in the Drebin dataset (ER $>$ 79%) but is robust against the malware examples in the Androzoo dataset (ER $<$ 1%). This finding indicates that the defense strategies of these COTS malware detectors are temporally evolving.

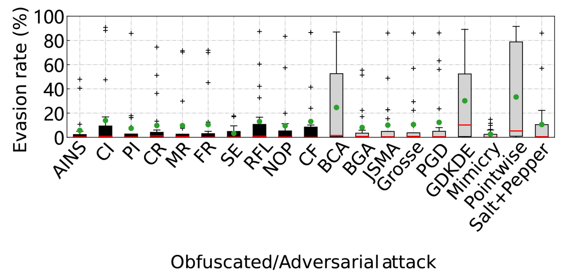
Finding 5: Evasion effectiveness of obfuscation and adversarial attacks against COTS malware detectors evolves with time.

We further inspect the evasion rates of each attack considering the Drebin dataset. Figure 3a depicts the evasion rates corresponding to the Drebin dataset and the 19 attacks against the 20 COTS malware detectors. We observe that the average evasion rate of the obfuscation attacks ranges from 19.91% to 45.04%. More specifically, SE is the most evasive, with an average evasion rate of 45.04% and a median value of 32.67%; NOP achieves an average evasion rate of 36.79%. For the adversarial attacks, GDKDE and Mimicry outperform the other attacks by achieving an average evasion rate of 37.89% and 37.58%, respectively.

Figure 3b presents the evasion rates of the same obfuscated examples and adversarial malware examples to the same 20 COTS malware detectors two months later. We then resubmit these examples to VirusTotal to observe if detectors are adaptive to attacks over time. We observe that these COTS malware detectors can be adaptive to obfuscated and adversarial malware examples. Specifically, the evasion rates of 16 attacks



(a) Result of the first submission



(b) Result of the second submission after two months

Fig. 3 Drebin dataset: the evasion rates of 19 attacks (i.e., 10 obfuscation attacks and 9 adversarial example attacks) on the 20 COTS malware detectors, where a black box indicates one obfuscation attack, a gray box indicates an adversarial attack, a red line denotes the median value, and a green circle indicates the mean value.

are markedly reduced two months later; for example, the average evasion rate of the SE attack drops to 3.20%, while its 3rd quantile value decreases from 88.41% to 4.83%. Meanwhile, the average evasion rate of the Mimicry attack decreases to 2.57% while its 3rd quantile value decreases from 84.70% to 2.45%. By contrast, three adversarial attacks, namely BCA, GDKDE, and Pointwise, still achieve average evasion rate of 26.64%, 30.75%, and 36.56%, respectively, while their 3rd quantile values are larger than 50%. Thus, adversarial attacks are more robust than obfuscation attacks considering their evasion capabilities against COTS malware detectors.

Finding 6: Among all of these attacks, SE obfuscation is the most evasive and then two adversarial attacks (i.e., GDKDE and Mimicry).

4.4 Answering RQ2

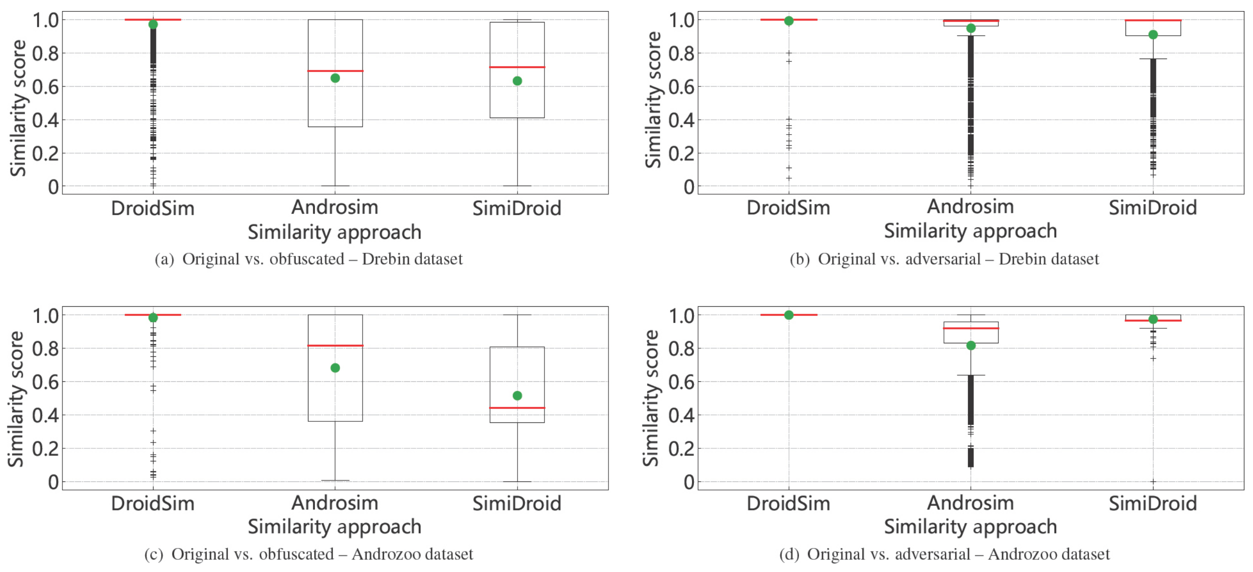
We compare original malware examples with their obfuscated version and adversarial examples by applying the three similarity-based approaches reviewed above (i.e., DroidSim, Androsim, and SimiDroid). Figure 4 depicts the distribution of similarity scores.

First, DroidSim shows that the obfuscated and adversarial examples are similar to their respective original examples. For obfuscated examples, the mean similarity scores are 0.97 and 0.98 on the Drebin and Androzoo dataset, respectively; for adversarial examples, the mean value is similarity score 0.99 for both datasets. Nevertheless, using MWW, we fail to reject the null hypothesis that the obfuscation attacks affect DroidSim differently from adversarial example attacks ($p = 9.52\times 10^{-262}$ and $p = 0$ on the Drebin and Androzoo datasets, respectively).

The Cliff's Delta presents a negligible difference, mainly due to the CF obfuscation attack. CF obfuscation can slightly affect DroidSim, with an average similarity of 0.83 on both datasets. This finding is due to the insertion of `goto` instructions to dexcode by CF, thereby changing the CB-CFG representation. For example, Fig. 5 shows how CF changes a subgraph of app 0A3BE4 (with an SHA256 checksum of 0a3b∗∗∗∗397b) by adding two nodes and edges.

Second, the distribution plot depicts that obfuscation attacks affect the Androsim approach worse than adversarial attacks. The mean and median similarity scores for obfuscated examples are 0.65 and 0.69 on the Drebin dataset, while those on the Androzoo dataset are 0.68 and 0.81, respectively. Meanwhile, the mean and median similarity scores for adversarial attacks are 0.95 and 0.99 on the Drebin dataset, while those on the Androzoo dataset are 0.82 and 0.83, respectively.

The MWW test showed that Androsim obtained different scores when applying obfuscation and adversarial attacks on both datasets ($p = 0$ on the Drebin dataset; $p = 4.50\times 10^{-233}$ on the Androzoo dataset). Cliff's Delta suggests that the difference is both medium on the Drebin and Androzoo datasets. This finding can be attributed to the vulnerability of Androsim to a large number of perturbations in methods; the similarity scores of AINS, CI, and NOP are lower than 0.52 on average. BGA markedly perturbs original malware examples and achieves 0.63 and 0.17 similarity scores on the Drebin and Androzoo datasets, respectively.



(a) Original vs. obfuscated – Drebin dataset
(b) Original vs. adversarial – Drebin dataset
(c) Original vs. obfuscated – Androzoo dataset
(d) Original vs. adversarial – Androzoo dataset

Fig. 4 Distribution of similarity scores given by DroidSim, Androsim, and SimiDroid.

The other adversarial attacks slightly perturb original malware examples, and the similarity scores are all larger than 0.8.

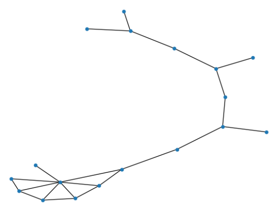
Third, the distribution plot depicts that obfuscation attacks affect the SimiDroid approach worse than adversarial attacks, and Cliff's Delta indicates large differences between these attacks on both datasets. This finding is mainly attributed to the key-mapping principle of SimiDroid, which increase the sensitivity of the SimiDroid approach to CR and MR; for example, CR obfuscation causes 0 similarities on both datasets. By contrast, the adversarial attacks barely rename app components despite being allowed to perform such an approach. In particular, the adversarial attacks learn small perturbations for evasion from the surrogate model (i.e., the Drebin model). The learned perturbations may be effective when thwarting local representation for malware detection, but they are substantially less sufficient to disturb the global expression of an app.

Finding 7: Obfuscation attacks affect code similarity detection approaches more than adversarial attacks owing to a large degree of perturbations of obfuscation attacks. Adversarial attacks are less capable of affecting the global representation of apps.

4.5 Answering RQ3

To answer RQ3, we install and run the original, obfuscated, and adversarial malware examples on an Android emulator to observe whether the original functionality of a malware example is changed by the attacks. We randomly selected 100 malware examples from the test set of the Drebin dataset. We use these examples to generate 1000 obfuscated examples and 900 adversarial examples. We install and run these examples on an Android emulator[✠], which is built on Android API version 6.0.1. We automatically run the examples by Monkey test. Similar to Ref. [23], each example is executed with 1000 events as input. We set the same seed value for each pair of examples: an original malware example and its manipulated variant. Because the seed value decides the randomness of the Monkey test (i.e., the same seed value leads to the same sequences of events). We use the command tool `logcat` to capture and store the logs during runtime.

Table 7 summarizes the experimental results, obtaining the following four observations. (1) All of the obfuscated and adversarial examples can be installed successfully. (2) Most attacks can produce runnable malware examples; for instance, 83 of the 100 AINS-incurred examples are runnable. Whereas, some attacks produce few runnable examples; for instance, 19 out of the 100 BGA-incurred examples are runnable. We randomly check the runtime log of 10 BGA-incurred examples. "RuntimeException" occurs to 7 of the examples by failing in instantiating activities. The 3 other examples crash due to *VerifyError* caused by a bad method. (3) BCA, JSMA, and NOP generate examples that behave similarly to the original version; this phenomenon can be attributed to the insertion-only manipulations. (4) The experimental results of Mimicry, GDKDE, CR, and MR attacks might be inaccurate owing to the renaming transformations. For instance, we compare the logs of the app B9C2A7 (with an SHA256


(a) Original malware example

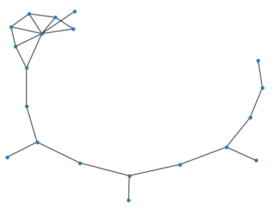

(b) Obfuscated example

Fig. 5 DroidSim subgraphs of an original malware example and its obfuscated example generated by the Control Flow (CF) attack.

Table 7 Number of examined, installable, runnable, and semantics-preserved malware examples generated by 10 obfuscation attacks and 9 adversarial attacks.

Attack	Examined	Installable	Runnable	Semantics-preserved
BCA	100	100	86	79
BGA	100	100	19	19
JSMA	100	100	75	71
Grosse	100	100	87	75
PGD	100	100	75	71
GDKDE	100	100	81	29
Mimicry	100	100	77	28
Pointwise	100	100	80	56
Salt+Pepper	100	100	82	77
AINS	100	100	85	83
CI	100	100	82	66
PI	100	100	84	80
CR	100	100	70	17
MR	100	100	83	70
FR	100	100	82	77
SE	100	100	82	62
RFL	100	100	85	73
NOP	100	100	83	73
CF	100	100	86	82

✠ https://mumu.163.com/.

checksum of b9c2∗∗∗∗9189) and the Mimicry-incurred version, and find that they display the same activities but with different class names.

Finding 8: Obfuscated malware examples generally preserve the functionality of their original malware examples, similarly to adversarial examples.

5 Discussion

This study aims to investigate the impact of adversarial attacks from three aspects empirically (i.e., malware detection, code similarity detection, and functionality preservation) by comparing them with the impact of obfuscation attacks. Through RQ1, our study provides some suggestions for the COTS malware detector vendors to improve COTS malware detectors in general. Obfuscation and adversarial attacks evade COTS malware detectors with approximately 30% evasion rates on average (Finding 1). Among all the attacks, two obfuscation attacks (i.e., SE and NOP) and two adversarial attacks (i.e., GDKDE and Mimicry) are the most effective (Finding 6). Besides, COTS malware detectors that are robust against obfuscation attacks can also defend against adversarial attacks (Finding 2). However, additional attention to adversarial attacks is required because they can explore more perturbations than obfuscation attacks, resulting in slightly higher evasion rates (Finding 3). Moreover, the robustness of some COTS malware detectors is temporal-based. Some detectors are robustness against attacks considering old examples, while some are vulnerable considering new examples (Finding 5). Both attacks may also raise suspicion of COTS malware detectors (Finding 4), which may be leveraged by the developers to improve the detection performance. Through RQ2, our study provides researchers additional sights to inspect evasion attacks. On the one hand, the code similarity detection approaches must provide more attention to obfuscation attacks than adversarial attacks; on the other hand, clustering can be leveraged to design future malware detectors against adversarial attacks (Finding 7). Through RQ3, obfuscation and adversarial attacks preserve functionality to a similar extent (Finding 8). Thus, both attacks must manipulate apps carefully, especially when using renaming transformations.

6 Threats to Validity

We now discuss the threats to the external, internal, and construct validity of the present study.

First, external validity means the extent to which our results are applicable to other settings. Android malware examples are evolving, and the findings may be specific to malware example datasets. Therefore, considering the aforementioned issue, we employ two datasets, which were collected during two different periods (i.e., August 2010–October 2012 for Drebin; July 2019–Dec 2019 for Androzoo). We observe different results from the two datasets. For instance, SymantecMobileInsight is robust against both obfuscated and adversarial examples in the Drebin dataset but is vulnerable to their counterparts from the Androzoo dataset. The external validity of our study may also be limited by our use of only 2870 original malware examples, which is slightly small. Future research should consider large datasets.

Second, internal validity refers to the effectiveness of the experimental results in supporting the obtained findings. For example, our generation of adversarial examples may not be optimal. Thus, we tune the hyperparameters of adversarial examples to optimize the attack effectiveness against the surrogate models to mitigate this issue.

Third, construct validity refers to the effectiveness of measurements in meeting the theoretical requirements for quantification. We measure semantics preservation according to display activities and exception behaviors because users often provide considerable attention to the UI and may not notice differences in the background. Two programs generally have the same semantics. However, defining additional quantitative metrics to measure the semantic similarity of the two programs might be possible.

7 Conclusion

We empirically quantify the difference between obfuscated malware examples and adversarial malware variants, both resulting from the same original malware examples. The following findings are presented. (1) Adversarial malware examples can evade COTS malware detectors slightly more often than their obfuscated malware examples. (2) Obfuscated malware examples negatively affect the code similarity detection approaches more than adversarial malware examples. (3) Obfuscation attacks and adversarial examples attacks preserve the functionality of original malware examples to a similar extent. We hope this study will motivate additional studies of defending against malware attacks, especially in the context of evasion attacks.

Appendix

Accuracy of COTS Malware Detectors

Table A1 represents the accuracy of each COTS malware detection to detect the original malware on the Drebin and Androzoo datasets. Most detectors work well in detecting Drebin malware examples because these examples are old and widely studied. By contrast, only a few COTS detectors can distinguish the malware in the Androzoo dataset because the examples are new and much less studied.

Table A1 Accuracy of selected COTS malware detectors to identify original malware examples.

(%)

Detector	Drebin dataset	Androzoo dataset
AVG	86.53	100.00
Ikarus	99.35	99.40
Fortinet	98.74	99.26
ESET-NOD32	98.19	98.80
Jiangmin	93.76	95.61
K7GW	99.10	89.66
SymantecMobileInsight	99.72	89.06
MaxSecure	62.04	75.01
Trustlook	95.93	55.15
Zillya	38.28	52.35
Antiy-AVL	98.92	50.27
CAT-QuickHeal	99.91	46.55
Cyren	98.47	23.06
BitDefenderFalx	99.39	10.16
Symantec	98.91	9.77
McAfee-GW-Edition	99.91	5.64
McAfee	99.64	3.78
MAX	99.10	3.55
AhnLab-V3	98.74	3.35
Cynet	98.00	3.26

References

[1] Y. Zhang, K. Wang, Q. He, F. Chen, S. Deng, Z. Zheng, and Y. Yang, Covering-based web service quality prediction via neighborhood-aware matrix factorization, *IEEE Trans. Serv. Comput.*, vol. 14, no. 5, pp. 1333–1344, 2021.

[2] H. Dai, J. Yu, M. Li, W. Wang, A. X. Liu, J. Ma, L. Qi, and G. Chen, Bloom filter with noisy coding framework for multi-set membership testing, *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 7, pp. 6710–6724, 2023.

[3] S. Wu, S. Shen, X. Xu, Y. Chen, X. Zhou, D. Liu, X. Xue, and L. Qi, Popularity-aware and diverse web APIs recommendation based on correlation graph, *IEEE Trans. Comput. Soc. Syst.*, vol. 10, no. 2, pp. 771–782, 2023.

[4] J. Zhou, M. Zhang, J. Sun, T. Wang, X. Zhou, and S. Hu, DRHEFT: Deadline-constrained reliability-aware HEFT algorithm for real-time heterogeneous MPSoC systems, *IEEE Trans. Rel.*, vol. 71, no. 1, pp. 178–189, 2022.

[5] Q. Wang, C. Zhu, Y. Zhang, H. Zhong, J. Zhong, and V. S. Sheng, Short text topic learning using heterogeneous information network, *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 5, pp. 5269–2581, 2023.

[6] Y. Zhang, G. Cui, S. Deng, F. Chen, Y. Wang, and Q. He, Efficient query of quality correlation for service composition, *IEEE Trans. Serv. Comput.*, vol. 14, no. 3, pp. 695–709, 2021.

[7] L. Qi, Y. Liu, Y. Zhang, X. Xu, M. Bilal, and H. Song, Privacy-aware point-of-interest category recommendation in internet of things, *IEEE Internet Things J.*, vol. 9, no. 21, pp. 21398–21408, 2022.

[8] Y. Liu, H. Wu, K. Rezaee, M. R. Khosravi, O. I. Khalaf, A. A. Khan, D. Ramesh, and L. Qi, Interaction-enhanced and time-aware graph convolutional network for successive point-of-interest recommendation in traveling enterprises, *IEEE Trans. Ind. Inform.*, vol. 19, no. 1, pp. 635–643, 2023.

[9] D. Zhou, X. Xue, and Z. Zhou, SLE2: The improved social learning evolution model of cloud manufacturing service ecosystem, *IEEE Trans. Ind. Inform.*, vol. 18, no. 12, pp. 9017–9026, 2022.

[10] X. Xue, S. Wang, L. Zhang, Z. Feng, and Y. Guo, Social learning evolution (SLE): Computational experiment-based modeling framework of social manufacturing, *IEEE Trans. Ind. Inform.*, vol. 15, no. 6, pp. 3343–3355, 2019.

[11] J. Zhou, L. Li, A. Vajdi, X. Zhou, and Z. Wu, Temperature-constrained reliability optimization of industrial cyber-physical systems using machine learning and feedback control, *IEEE Trans. Automat. Sci. Eng.*, vol. 20, no. 1, pp. 20–31, 2023.

[12] R. Gu, Y. Chen, S. Liu, H. Dai, G. Chen, K. Zhang, Y. Che, and Y. Huang, Liquid: Intelligent resource estimation and network-efficient scheduling for deep learning jobs on distributed GPU clusters, *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2808–2820, 2022.

[13] H. Dai, C. Wu, X. Wang, W. Dou, and Y. Liu, Placing wireless chargers with limited mobility, in *Proc. the IEEE INFOCOM 2020 – IEEE Conf. Computer Communications*, Toronto, Canada, 2020, pp. 2056–2065.

[14] J. Zhou, K. Cao, X. Zhou, M. Chen, T. Wei, and S. Hu, Throughput-conscious energy allocation and reliability-aware task assignment for renewable powered *in-situ* server systems, *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 41, no. 3, pp. 516–529, 2022.

[15] J. Gao, X. Liu, Y. Chen, and F. Xiong, MHGCN: Multiview highway graph convolutional network for cross-lingual entity alignment, *Tsinghua Science and Technology*, vol. 27, no. 4, pp. 719–728, 2022.

[16] Y. Yang, X. Yang, M. Heidari, M. A. Khan, G. Srivastava, M. Khosravi, and L. Qi, ASTREAM: Data-stream-driven scalable anomaly detection with accuracy guarantee in IIoT environment, *IEEE Trans. Netw. Sci. Eng.*, doi: 10.1109/TNSE.2022.3157730.

[17] L. Qi, Y. Yang, X. Zhou, W. Rafique, and J. Ma, Fast anomaly identification based on multiaspect data streams for intelligent intrusion detection toward secure industry 4.0, *IEEE Trans. Ind. Inform.*, vol. 18, no. 9, pp. 6503–6511, 2022.

[18] F. Wang, G. Li, Y. Wang, W. Rafique, M. R. Khosravi, G. Liu, Y. Liu, and L. Qi, Privacy-aware traffic flow prediction based on multi-party sensor data with zero trust in smart city, *ACM Trans. Internet Technol.*, https://doi.org/10.1145/3511904 , 2022.

[19] Y. Zhang, J. Pan, L. Qi, and Q. He, Privacy-preserving quality prediction for edge-based IoT services, *Future Gener. Comput. Syst.*, vol. 114, pp. 336–348, 2021.

[20] E. Shein, Malware is down, but IoT and ransomware attacks are up, https://www.techradar.com/news/iot-malware-attacks-saw-a-huge-rise-last-year, 2020.

[21] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, Drebin: Effective and explainable detection of android malware in your pocket, in *Proc. 21st Annu. Network and Distributed System Security Symposium* (*NDSS*), San Diego, CA, USA, doi: 10.14722/ndss.2014.23247.

[22] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, TESSERACT: Eliminating experimental bias in malware classification across space and time, in *Proc. 28th USENIX Security Symp.*, Santa Clara, CA, USA, 2019, pp. 729–746.

[23] M. Hammad, J. Garcia, and S. Malek, A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products, in *Proc. 40th Int. Conf. Software Engineering*, Gothenburg, Sweden, 2018, pp. 421–431.

[24] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, Intriguing properties of adversarial ML attacks in the problem space, in *Proc. 2020 IEEE Symp. Security and Privacy*, San Francisco, CA, USA, 2020, pp. 1332–1349.

[25] J. Jung, C. Jeon, M. Wolotsky, I. Yun, and T. Kim, AVPASS: Leaking and bypassing antivirus detection model automatically, https://github.com/sslab-gatech/avpass, 2022.

[26] S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo, Obfuscapk: An open-source black-box obfuscation tool for Android apps, *SoftwareX*, vol. 11, p. 100403, 2020.

[27] M. Saleh, E. P. Ratazzi, and S. Xu, Instructions-based detection of sophisticated obfuscation and packing, in *Proc. 2014 IEEE Military Communications Conf.*, Baltimore, MD, USA, 2014, pp. 1–6.

[28] M. Zheng, P. P. C. Lee, and J. C. S. Lui, ADAM: An automatic and extensible platform to stress test android anti-virus systems, in *Proc. 9th Int. Conf. Detection of Intrusions and Malware*, and Vulnerability Assessment, Heraklion, Greece, 2012, pp. 82–101.

[29] V. Rastogi, Y. Chen, and X. Jiang, DroidChameleon: Evaluating android anti-malware against transformation attacks, in *Proc. 8th ACM SIGSAC Symp. Information, Computer and Communications Security*, Hangzhou, China, 2013, pp. 329–334.

[30] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, Stealth attacks: An extended insight into the obfuscation effects on Android malware, *Comput. Secur.*, vol. 51, pp. 16–31, 2015.

[31] D. Li, Q. Li, Y. F. Ye, and S. Xu, Arms race in adversarial malware detection: A survey, *ACM Comput. Surv.*, vol. 55, no. 1, p. 15, 2021.

[32] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, Adversarial examples for malware detection, in *Proc. 22nd European Symp. Research in Computer Security*, Oslo, Norway, 2017, pp. 62–79.

[33] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren, Android HIV: A study of repackaging malware for evading machine-learning detection, *IEEE Trans. Inform. Forensic. Secur.*, vol. 15, pp. 987–1001, 2020.

[34] L. Li, T. F. Bissyandé, and J. Klein, Rebooting research on detecting repackaged android apps: Literature review and benchmark, *IEEE Trans. Softw. Eng.*, vol. 47, no. 4, pp. 676–693, 2021.

[35] A. Al-Dujaili, A. Huang, E. Hemberg, and U. M. O'Reilly, Adversarial deep learning for robust detection of binary encoded malware, in *Proc. 2018 IEEE Security and Privacy Workshops* (*SPW*), San Francisco, CA, USA, 2018, pp. 76–82.

[36] D. Li and Q. Li, Adversarial deep ensemble: Evasion attacks and defenses for malware detection, *IEEE Trans. Inform. Forensic. Secur.*, vol. 15, pp. 3886–3900, 2020.

[37] Z. Kolter and A. Madry, Adversarial robustness: Theory and practice, https://adversarial-ml-tutorial.org/, 2021.

[38] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, The limitations of deep learning in adversarial settings, in *Proc. 2016 IEEE European Symp. Security and Privacy* (*EuroS&P*), Saarbrücken, Germany, 2016, pp. 372–387.

[39] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli, Evasion attacks against machine learning at test time, in *Proc. European Conf. Machine Learning and Knowledge Discovery in Databases*, Prague, Czech Republic, 2013, pp. 387–402.

[40] L. Schott, J. Rauber, M. Bethge, and W. Brendel, Towards the first adversarially robust neural network model on MNIST, in *Proc. 7th Int. Conf. Learning Representations* (*ICLR*), New Orleans, LA, USA, https://openreview.net/forum?id=S1EHOsC9tX, 2019.

[41] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli, Yes, machine learning can be more secure! A case study on android malware detection, *IEEE Trans. Depend. Secure Comput.*, vol. 16, no. 4, pp. 711–724, 2019.

[42] R. Gu, K. Zhang, Z. Xu, Y. Che, B. Fan, H. Hou, H. Dai, L. Yi, Y. Ding, G. Chen, and Y. Huang, Fluid: Dataset abstraction and elastic acceleration for cloud-native deep learning training jobs, in *Proc. 2022 IEEE 38th Int. Conf. Data Engineering* (*ICDE*), Kuala Lumpur, Malaysia, 2022, pp. 2182–2195.

[43] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, AndroZoo: Collecting millions of android apps for the research community, in *Proc. 2016 IEEE/ACM 13th Working Conf. Mining Software Repositories* (*MSR*), Austin, TX, USA, 2016, pp. 468–471.

[44] V. Sihag, M. Vardhan, and P. Singh, A survey of android application and malware hardening, *Comput. Sci. Rev.*, vol. 39, p. 100365, 2021.

[45] A. Kovacheva, Efficient code obfuscation for android, in

Proc. 6th Int. Conf. Advances in Information Technology, Bangkok, Thailand, 2013, pp. 104–119.

[46] V. Balachandran, Sufatrio, D. J. J. Tan, and V. L. L. Thing, Control flow obfuscation for Android applications, *Comput. Secur.*, vol. 61, pp. 72–93, 2016.

[47] L. Chen, S. Hou, and Y. Ye, SecureDroid: Enhancing security of machine learning-based detection against adversarial android malware attacks, in *Proc. 33rd Annu. Computer Security Applications Conf.*, Orlando, FL, USA, 2017, pp. 362–372.

[48] A. Desnos, Android: Static analysis using similarity distance, in *Proc. 2012 45th Hawaii Int. Conf. System Sciences*, Maui, HI, USA, 2012, pp. 5394–5403.

[49] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie, Detecting code reuse in android applications using component-based control flow graph, in *Proc. 29th IFIP TC 11 Int. Conf. ICT Systems Security and Privacy Protection*, Marrakech, Morocco, 2014, pp. 142–155.

[50] J. Park, H. Kim, Y. Jeong, S. Cho, S. Han, and M. Park, Effects of code obfuscation on android app similarity analysis, *J. Wirel. Mob. Netw. Ubiquit. Comput. Depend. Appl.*, vol. 6, no. 4, pp. 86–98, 2015.

[51] Y. Zhang, G. Xiao, Z. Zheng, T. Zhu, I. W. Tsang, and Y. Sui, An empirical study of code deobfuscations on detecting obfuscated android piggybacked apps, in *Proc. 2020 27th Asia-Pacific Software Engineering Conf.* (*APSEC*), Singapore, 2020, pp. 41–50.

[52] W. Brendel, J. Rauber, and M. Bethge, Decision-based adversarial attacks: Reliable attacks against black-box machine learning models, in *Proc. 6th Int. Conf. Learning Representations* (*ICLR*), Vancouver, Canada, https://openreview.net/forum?id=SyZI0GWCZ, 2018.

[53] L. Li, T. F. Bissyandé, and J. Klein, SimiDroid: Identifying and explaining similarities in android apps, in *Proc. 2017 IEEE Trustcom/BigDataSE/ICESS*, Sydney, Australia, 2017, pp. 136–143.

[54] R. Cilibrasi and P. M. Vitanyi, Clustering by compression, *IEEE Trans. Inform. Theory*, vol. 51, no. 4, pp. 1523–1545, 2005.

[55] H. Huang, Z. Zeng, D. Yao, X. Pei, and Y. Zhang, Spatial-temporal ConvLSTM for vehicle driving intention prediction, *Tsinghua Science and Technology*, vol. 27, no. 3, pp. 599–609, 2022.

Yan Xu received the BEng degree in software engineering from the Nanjing University of Science and Technology, China in 2017, where she is currently a PhD candidate in computer science and technology. Her research interests include malware detection and adversarial attacks.

Deqiang Li received the MEng degree in software engineering and the PhD degree in computer science and technology from Nanjing University of Science and Technology, China in 2017 and 2021, respectively. He is currently a lecturer at Nanjing University of Posts and Telecommunications. His research interests include adversarial malware detection, adversarial machine learning, and applied data mining techniques in malware detection.

Qianmu Li received the BEng and PhD degrees in computer application technology from Nanjing University of Science and Technology, China in 2001 and 2005, respectively. He worked as a postdoctoral researcher at Nanjing University from 2005 to 2007. He is currently a full professor and member of the Academic Committee at Nanjing University of Science and Technology, the director of informatization division; the vice chairman of Jiangsu Provincial Science Association. His research interests include big data analysis, cyberspace security, and software systems. He has published more than 110 scientific articles and received many research grants from China's national and provincial programs. He has received many Best Paper Awards from ISKE, AAAI, ICCC, EAI, etc.

Shouhuai Xu received the PhD degree in computer science from Fudan University, China in 2000. He is the Gallogly Chair Professor in cybersecurity at the Department of Computer Science, University of Colorado Colorado Springs (UCCS), USA. He pioneered the cybersecurity dynamics approach as foundation for the emerging science of cybersecurity, with three pillars: first-principle cybersecurity modeling and analysis (the x-axis); cybersecurity data analytics (the y-axis, to which the present paper belongs); and cybersecurity metrics (the z-axis). He co-initiated the International Conference on Science of Cyber Security and is serving as its steering committee chair. He is/was an associate editor of *IEEE Transactions on Dependable and Secure Computing* (*IEEE TDSC*), *IEEE Transactions on Information Forensics and Security* (*IEEE T-IFS*), *IEEE Transactions on Network Science and Engineering* (*IEEE TNSE*), and *Scientific Reports*. More information about his research can be found at https://xu-lab.org.