information

Article

# On the Feasibility of Adversarial Sample Creation Using the Android System API

**Fabrizio Cara** *, **Michele Scalas**, **Giorgio Giacinto and Davide Maiorca**

Department of Electrical and Electronic Engineering, University of Cagliari, 09123 Cagliari, Italy; michele.scalas@unica.it (M.S.); giacinto@unica.it (G.G.); davide.maiorca@unica.it (D.M.)

* Correspondence: fabrizio.cara@unica.it

Received: 31 July 2020; Accepted: 7 September 2020; Published: 10 September 2020

**Abstract:** Due to its popularity, the Android operating system is a critical target for malware attacks. Multiple security efforts have been made on the design of malware detection systems to identify potentially harmful applications. In this sense, machine learning-based systems, leveraging both static and dynamic analysis, have been increasingly adopted to discriminate between legitimate and malicious samples due to their capability of identifying novel variants of malware samples. At the same time, attackers have been developing several techniques to evade such systems, such as the generation of evasive apps, i.e., carefully-perturbed samples that can be classified as legitimate by the classifiers. Previous work has shown the vulnerability of detection systems to evasion attacks, including those designed for Android malware detection. However, most works neglected to bring the evasive attacks onto the so-called problem space, i.e., by generating concrete Android adversarial samples, which requires preserving the app's semantics and being realistic for human expert analysis. In this work, we aim to understand the feasibility of generating adversarial samples specifically through the injection of system API calls, which are typical discriminating characteristics for malware detectors. We perform our analysis on a state-of-the-art ransomware detector that employs the occurrence of system API calls as features of its machine learning algorithm. In particular, we discuss the constraints that are necessary to generate real samples, and we use techniques inherited from interpretability to assess the impact of specific API calls to evasion. We assess the vulnerability of such a detector against mimicry and random noise attacks. Finally, we propose a basic implementation to generate concrete and working adversarial samples. The attained results suggest that injecting system API calls could be a viable strategy for attackers to generate concrete adversarial samples. However, we point out the low suitability of mimicry attacks and the necessity to build more sophisticated evasion attacks.

**Keywords:** Android; malware detection; adversarial machine learning; evasion attack; problem space

## 1. Introduction

Android is the most used and attacked mobile operating system. According to McAfee [1], the number of total malicious applications detected on mobile devices has continuously grown through the four quarters of 2019. As Kaspersky reported [2], nineteen entries of the top 20 mobile malware applications for the year 2019 target Android. For this reason, multiple efforts have been made on the design of malware detection systems to identify potentially harmful applications. All these security solutions employ different strategies based on the analysis of the application through static analysis [3–5], dynamic analysis [6–8], or a combination of them. Additionally, such information is often employed by machine learning algorithms to carry out an accurate detection of known and previously unseen attacks [9–13]. However, machine learning algorithms are vulnerable to well-crafted attacks. In fact, attackers have developed several techniques to evade such systems, ranging from code obfuscation to adversarial attacks, i.e., modifications to the samples that directly target learning algorithms such that input samples are misclassified.

Previous work has extensively shown the vulnerability of learning-based detection systems, including those designed for Android malware detection ([14,15]), to test-time evasion attacks, which consist of creating carefully-perturbed malicious samples that are able to be classified as legitimate by the classifiers. However, a critical problem that has been often overlooked in previous work is the practical feasibility of generating adversarial samples. When designing a machine learning-based detection system, experts identify a set of characteristics (features) that are expected to be effective in classifying the input samples. That is, they create a feature space where they map specific characteristics, patterns, or behaviors of the sample to a feature vector. Conversely, when performing adversarial attacks, attackers generate an altered feature vector, thus converting each alteration into a modification of the samples (in the problem space) in order to attain the desired evasion. Depending on the setting, this transition is not straightforward and entails the difficulty of moving from the feature space to the problem space (or vice versa), i.e., finding an exact, unique correspondence between values in the feature vector and characteristics in the problem domain (e.g., functionalities in an Android application). This is the so-called inverse feature-mapping problem [15–17]—or the more generic problem-feature space dilemma [18]. Moreover, the generation of concrete, realistic adversarial samples also requires taking into account different constraints, such as preserving the app's semantics or keeping it plausible for human inspection [19,20].

In this work, we explore the development of evasive Android apps in the problem space. In particular, we aim to understand the feasibility of generating such samples specifically through the injection of system API calls, which are known to be discriminative features for malware detectors. We analyze an injection strategy that only adds the calls needed to achieve the evasion by preserving the application's overall functionality. Consequently, we consider a scenario where both the attacker's effort and the impact on the generated app are kept at a minimum level. Moreover, different from all previous articles in the literature, we evaluate our strategy on a detector designed with non-binary features. Overall, the main contributions of this paper are the following:

- We discuss the constraints required to create concrete, working Android adversarial samples through API call injection;
- We evaluate the feasibility of injecting system API calls by both identifying the subset of the usable ones and explaining their relevance to evasion through a gradient-based interpretability technique;
- We evaluate the effectiveness of mimicry and random noise addition attacks against a state-of-the-art ransomware detector that employs non-binary features;
- We develop a basic implementation of the considered injection strategy that creates working adversarial malicious samples.

We believe that this first proposal starts to highlight what viable injection strategies attackers have at their disposal for the creation of real evasive malware samples.

The paper is organized as follows. In Section 2, we present the background knowledge about Android, malware detection, and its evasion. Section 3 proposes an overview of the literature about the creation of adversarial samples in the problem space. In Section 4, we illustrate the proposed methodology. In particular, we describe the threat model of our setting. Then, we discuss the problem space domain with its constraints. Finally, we describe our implementation for the generation of Android adversarial samples. In Section 5, we present the experiments that we conducted. Section 6 closes the paper with the conclusive remarks, along with a discussion of its limitations and future work.

## 2. Android Background

In this section, we first present how Android applications are structured, primarily focusing on the elements containing the apps' instructions, which are the `.dex` files. Then, we discuss Android malware detection, including machine learning-based systems. Finally, we describe how it is possible to elude such detectors.

### *2.1. Android Applications*

Android applications are well-organized structures whose elements are contained in a single file. This file is a compressed archive whose extension is `.apk` (i.e., Android application package). In particular, an Android application is composed of the following elements:

- `AndroidManifest.xml`: This specifies the structure and the main components of the application; for example, it lists the permissions required by the app and its activities, i.e., the components that usually show a user interface;
- Resources : graphical elements and `.xml` files used to define the layout properties of the application;
- Assets: external resources of the application, such as multimedia files and native libraries;
- `Classes.dex`: *dex* stands for Dalvik EXecutable. Android apps have one or more of these files, which contain the executable code of the application. Since our work focuses on the alteration of `dex` code, we describe the content of `.dex` files in the following.

Android applications are written in Java or Kotlin. Then, in both cases, the code is compiled into Dalvik bytecode, which is contained in the `.dex` files (in the majority of the cases, there is a unique `.dex` file called `classes.dex`). This bytecode can be further disassembled into `Smali`, a human-readable format. From Android 4.4 onwards, the bytecode is converted to native ARM code during the installation of the application (ahead-of-time approach), and it is then executed by Android Runtime (ART). A `.dex` file is a layered container of bytes organized by reference. Therefore, class definitions contain all the information as a reference to a data structure contained in the `.dex` file. The main components of a `.dex` file are:

- Header: This contains information about the file composition, such as the offsets and the size of other parts of the file (such as constants and data structures). This data collection is crucial to reconstruct the bytecode in the correct way when the code is compiled to ARM;
- Constants: They represent the addresses of the strings, flags, variables, classes, and method names of the application;
- Classes: This is the definition of all the class parameters, like the superclass, the access type, and the list of methods with all the references to the data contained in the data structure;
- Data structure: This is the container for the actual data of the application, such as the method code or the content of static variables.

To better understand `.dex` files, it is possible to imagine them as a sequence of hierarchical references that, starting from the header, lead to the target data structures.

In this work, particular attention is given to the Dalvik bytecode instructions to call a method. These are called `invoke`. As shown in Figure 1, an `invoke`-type instruction can be direct (to call a constructor, a static, or a private method) or virtual (to call a public method); it may have multiple registers in which the method parameters are contained, and it features the class and method to call (as well as the return type of the method).

invoke-*type* {vC, vD, vE, vF, vG}, *class->method()returnType*

**Figure 1.** General form of an invoke instruction.

### *2.2. Android Malware Detection*

Mobile attacks are a real security issue for Android users, as a wide variety of malware families has been released in the wild, each one with different peculiarities. Two different analyses and detection approaches are generally used to mitigate this issue: static analysis and dynamic analysis.

Static analysis is based on disassembling an Android application and scanning its components to find malicious traces without executing the application. A malicious trace is a sequence of bytes (which can be translated, for example, to a sequence of instructions) with which it is possible to recognize a particular malware family. Concerning Android, different research works have been published about this type of analysis. Feng et al. [3] proposed Apposcopy, a detection tool that combines static taint analysis and intent flow monitoring to produce a signature for applications. Arzt et al. [4] proposed FlowDroid, a security tool that performs static taint analysis within the single components of Android applications. Zhou et al. [5] proposed DroidRenger, a detection system that performs static analysis on previously unknown Android malicious applications in official and third-party market stores. Static analysis is quick, and it requires low computational resources and time. For this reason, it can be implemented on Android devices as well. However, this technique is subject to high rates of false positives. The reason is that to perform static analysis, it is necessary to know a malicious trace in advance, so this detection method may be easily evaded by obfuscating the code.

Concerning dynamic analysis, it is based on executing and monitoring an application in a controlled environment (i.e., sandbox or virtual machine). The goal is to inspect the interactions between the application and the system to reveal all the suspicious behaviors. With respect to Android, multiple systems have been proposed about this type of analysis. Tam et al. [6] proposed CopperDroid, a dynamic analyzer that aims to identify suspicious high-level behaviors of malicious Android applications. Zhang et al. [7] proposed VetDroid, a dynamic analysis platform to detect interactions between the application and the system through the monitoring of permission use behaviors. Chen et al. [8] proposed RansomProber, a dynamic ransomware detection system that uses a collection of rules to examine several execution aspects, such as the presence of encryption or unusual layout structures. Dynamic analysis is more challenging to implement. It requires more computational resources and time to be executed. This is why this type of analysis cannot be implemented on a mobile device. However, it has better performances in detecting well-known and never seen malware families.

Currently, machine learning-based systems have become widely employed to detect whether or not an application is malicious. This is because this type of algorithms can identify particular patterns by analyzing specific application behaviors, such as resource usage, system calls, and specific permissions. A pattern is defined as a collection of features that may describe a particular behavior in a unique way. Hence, once the malicious patterns are defined, it is possible to identify all the applications that fall into those patterns and classify them as malicious. Generally, the features used for the classification of patterns are gathered using static or dynamic analysis.

**R-PackDroid [12,21].** We now briefly describe the work by Maiorca et al. [12], which is the basis of the setting considered in this paper. The authors proposed a system designed for the detection of Android ransomware attacks. This machine learning-based system uses three labels to classify Android applications: benign, malicious, and ransomware. The feature set consists of the occurrence of API package calls, which marks a difference from other malware detection proposals, such as DREBIN [10], which considers the simple presence of specific calls (system-related or not), resulting in a binary feature vector. This system has attained high performance in discriminating ransomware samples. On the basis of this work, Scalas et al. [21] performed a comparison of the impact of more granular features (API class calls and API method calls) on detection. In our case, we consider the original proposal of API package calls.

### *2.3. Detection Evasion and Defense*

Although the detection approaches discussed in Section 2.2 help improve Android's security by detecting malicious activities, they are far from being infallible. Different techniques may be employed to evade detection.

When static analysis is employed to detect malicious behaviors, it is possible to modify the application to hide the malicious content by using obfuscation techniques. As multiple research works have shown [22,23], these techniques may be differentiated into two primary groups: trivial and non-trivial. Trivial obfuscation techniques are easy to implement, and they are based on modifying the structural application files without altering the code. The non-trivial ones are more challenging to implement, and they are based on the modification of the structural application files and code, thus resulting in a more effective detection elusion.

To detect and analyze the aforementioned obfuscation techniques, it is possible to use specific detection tools to see if an application has any signs of trivial or non-trivial obfuscation [24]. Another possible way to analyze an obfuscated application is through dynamic analysis, which allows the analyst to access the content of the variables and the obfuscated code's output at run-time.

Since dynamic analysis is usually performed inside a virtual environment (to preserve the system from malicious executions), it is possible to employ a tool to detect virtualization. Hence, attackers may block the execution of some routines to hide the malicious behavior.

**Adversarial Machine Learning.** As pointed out by multiple research works [14,25,26], machine learning algorithms are tested with datasets whose samples have the same probability distribution. This means that machine learning classifiers that do not employ any adversary-aware approach are vulnerable to well-crafted attacks that violate this assumption. This scenario is called adversarial machine learning.

Among adversarial offensives, in evasion attacks, attackers take a malicious application and manipulate its features to lead the classifier to misclassify the malicious sample as a benign one. To do so, the features that are generally used are the system or non-system calls (classes and packages) and the information contained in the Android Manifest, such as, among others, permissions and activities. To increase the security of machine learning-based classifiers, the training phase of these systems is conducted proactively, trying to predict possible attacks with threat modeling techniques [14]. A proactive approach depends on the considered attack scenario, which is based on the system knowledge of the attacker. The higher is the attacker's system knowledge, the more effective is the attack.

## 3. Related Work

While it is possible to find multiple research efforts about the formal aspects of the evasion attack, few of them have focused on creating adversarial samples that accomplish this kind of attack in practice.

Concerning the Android domain, Pierazzi et al. [19] proposed a formalization for problem space attacks and a methodology to create evasive Android malware. In particular, starting from the feature space problem, which has been widely discussed in the literature, they identified the constraints to keep the generated samples working and realistic. This means creating adversarial samples that are robust to pre-processing analysis, preserved in its semantics, completely functioning, and feasible in its transformations. We discuss such constraints more extensively in Section 4.2.1. To generate the adversarial samples, they used an automated software transplantation technique, which consists of taking a piece of code (that contains the wanted features) from another application. Finally, they evaluated their system on DREBIN [10], an Android malware detection system based on an SVM algorithm that uses binary features, and its hardened variant, which uses a Sec-SVM classifier [14]. They showed that it is possible to create a sample that evades the classification (with about 100% probability), making about two dozen transformations for the SVM classifier and about a hundred transformations for the Sec-SVM classifier. While this work is robust and effective in its methodology, it could be possible to use an opaque predicate detection mechanism to detect the unreachable branches [27,28].

Grosse et al. [29] proposed a sample generation method with only one constraint: keeping the semantics of the evasive samples consistent with a maximum of 20 transformations. They tested their system with a deep neural network classifier trained using the DREBIN dataset. However, their work only performed minimal modifications to the `Manifest`, so these modifications may be detected with a static analysis tool made to remove unused permissions and undeclared classes. Furthermore, the adversarial samples were not tested, so there is no way to know whether the adversarial samples were correctly executed.

Yang et al. [30] proposed a methodology to create adversarial Android malware following two principal constraints: preserving the malicious behaviors and maintaining the robustness of the application (e.g., correct installation and execution). They evaluated their adversarial malware system against the DREBIN classifier [10] and the AppContext classifier [31]. However, their methodology lacks in preserving the stability of the application (e.g., they show high rates of adversarial application crashes), especially when the number of features to add increases.

It is worth noting that, different from our case (see Section 4), all the above-mentioned articles evaluated their proposals on systems with binary features, thus only highlighting the presence or absence of certain characteristics in the app.

For what concerns other domains, Song et al. [20] presented an open-source framework to create adversarial malware samples capable of evading detection. The proposed system firstly generates the adversarial samples with random modifications; then, it minimizes the sample, removing the useless features for the classification. They used two open-source classifiers: Ember and ClamAV. They also described the interpretation of the features to give a better explanation of why the generated adversarial samples evade the classification. They showed that the generated adversarial malicious samples can evade the classification and that, in some cases, the attacks are transferable between different detection systems.

Rosenberg et al. [32] proposed an adversarial attack against Windows malware classifiers based on API calls and static features (e.g., printable strings). They evaluated their system with different variants of RNN (recurrent neural network) and traditional machine learning classifiers. However, their methodology is based on the injection of no-op API calls that may be easily detected with a static code analyzer. Hu and Tan [33] proposed a generative adversarial network (GAN) to create adversarial malware samples. They evaluated their system on different machine learning-based classifiers. However, their methodology is not reliable because of the use of GAN, which is known to have an unstable training process [34].

## 4. Model Description And Methodology

In this section, we first describe the threat model to perform the evasion attack in the feature space (Section 4.1), followed by the constraints to consider in order to accomplish it in the problem space (Section 4.2). Finally, we illustrate the steps adopted to generate the concrete adversarial samples (Section 4.3).

### *4.1. Threat Model*

The typical main aspects to consider when modeling the adversarial threats are the following: the attacker's goal, the attacker's knowledge, and the attacker's capability [14,15,20].

**Attacker's goal.** This consists of pursuing an indiscriminate or targeted attack. In the first case, the attacker is generically interested in having the samples misclassified. In the second case, its goal is to have specific samples classified as a target class. In our setting, the attackers aim to make the detection system classify ransomware apps as trusted ones. In particular, since we consider the evasion strategy, they fulfill this goal by modifying the Android apps at test time rather than by poisoning the system's training set.

**Attacker's knowledge.** Given a knowledge parameter $\Phi$ that indicates the amount of information about the target detection system available to them, this can be related to:

*(a)* the dataset $\boldsymbol{D}$;
*(b)* the feature space $\boldsymbol{X}$;
*(c)* the classification function $\boldsymbol{f}$.

Accordingly, $\Phi = (\boldsymbol{D}, \boldsymbol{X}, \boldsymbol{f})$ corresponds to the scenario of perfect knowledge about the system and represents the worst case for a defender. However, it is unlikely to see such a scenario in real cases, as attackers often have incomplete information (or no information at all) about the target system. For this reason, in this work, we simulate a scenario where the attacker has minimum information about the Android detection system. Specifically, we focus on the mimicry attack, which has been studied in previous work [14–16]. In this case, $\Phi = (\hat{\boldsymbol{D}}, \boldsymbol{X})$, which means that the attacker knows the feature space and owns a set of data that is a representative approximation of the probability distribution of the data employed in the target system. This is a more realistic scenario, in which the attacker can modify the features of a malicious sample to make its feature vector as similar as possible to one of the benign samples at its disposal. As a comparison, we also consider a random noise addition attack, which does not allow targeting a specific class, but can be useful to provide a generic assessment of the vulnerability of the system to perturbed inputs.

**Attacker's capability** This refers to the kind of modifications that the attacker is able to perform on a sample, e.g., a malicious Android application. For example, the Android detection domain mostly allows the attacker to only add new elements to the app (feature addition), such as permissions, strings, or function calls, but does not permit removing them (feature removal). We discuss this aspect with more detail in the section that follows.

## *4.2. The Problem Space Domain*

Our goal is to evaluate to what extent it is feasible to generate real-world Android adversarial samples. In particular, in this work, we focus our analysis specifically on the constraints and consequences of injecting system API calls. To do so, the first concern to consider is the so-called inverse feature-mapping problem—or the more generic problem-feature space dilemma [18]. As hinted at in Section 1, this refers to the difficulty of moving from the feature space to the problem domain (or vice versa), i.e., finding an exact, unique correspondence between values in the feature vector and characteristics in the problem domain, e.g., functionalities in an Android application. The feature mapping problem can be defined as a function $\psi$ that, given a sample $z$, generates a $d$-dimensional feature vector $\boldsymbol{x} = [x_1, x_2, ..., x_d]$, such that $\psi(z) = \boldsymbol{x}$ [19]. Conversely, the opposite flux in the inverse feature-mapping case is a function $\mathcal{S}$, such that taking a feature vector $\boldsymbol{x}$, we have $\mathcal{S}(\boldsymbol{x}) = z'$. However, it is not guaranteed that $z \equiv z'$.

As an example, let us consider the feature vector of our setting, which consists of the occurrence of API package calls. Due to this choice, the value of a feature $x_i$ in the feature vector can be increased through two main behaviors in the Android application: (a) the call of a class constructor and (b) the call of a method. In both cases, the class involved in these calls must belong to the package that corresponds to the $i$-th feature. This means that there are as many ways to change that feature value as the number of callable classes in the package. Figure 2 exemplifies this aspect by showing, for a few packages, their related classes that we identify as callable for our purposes. By contrast, an alternative feature vector (as discussed in [21]) that describes the occurrence of system API method calls would have a one-to-one mapping between the $i$-th feature and the call of the corresponding method.

```
android.content -> ContentUris, ContentValues, Intent, IntentFilter
android.database -> ContentObservable, DatabaseUtils, DataSetObservable
android. graphics -> Picture, Regiorn, ColorFilter, Camera
```

**Figure 2.** List of usable classes for three different Android packages.

The above-described issue is particularly relevant for the creation process of adversarial samples. Another implication to consider is the potential presence of side-effect features, i.e., the undesired alteration of features besides the ones targeted in the attack [19]. For example, inserting whole portions of code to add specific calls may have the effect of injecting unnecessary, additional calls. This may lead to an evasive feature vector that is slightly different from the expected one, thus making the behavior of the target classifier unpredictable.

The injection approach considered in this work starts from the will of inserting the minimum amount of modifications needed to evade the detection. However, other concerns must be taken into account in order to create a realistic, working adversarial malware. We discuss them below.

### 4.2.1. Constraints

We now present the constraints that we consider in our approach and their implications on the injection strategy design. We illustrate them on top of the definitions proposed by Pierazzi et al. [19]. We refer the reader to that work for further details.

**Available transformations.** The modification of the features has to correspond to doable actions in the problem domain. That is, it is necessary to evaluate the set of possible transformations. In the case of Android, some sample modifications could lead to a change in the app behavior, a crash during the execution, or rejection by the Android Verifier. Typically, the attacker can only add new elements to the apps (feature addition), such as permissions, strings, or function calls, while it is harder for it to remove them (feature removal). For example, it is not possible to remove permissions from the `Manifest`.

In this work, we choose the feature addition strategy, as we only inject new system API calls into the `dex` code. In this sense, it is possible to successfully perform the modifications only for a reduced set of Android system packages and classes. In fact, the packages we inject are the ones whose classes are not interfaces or abstract classes and whose constructors are public and accessible. Another issue is related to the call parameters. These have to be correctly defined because Java has a strict, static type checking. Thus, to call methods or constructors that receive specific parameters, one could create and pass new objects of the needed classes. Since this can result in being a complicated procedure, in this work, we start exploring the most straightforward setting for attackers, i.e., where they restrict the set of callable classes to the ones that need primitive or no parameters at all. We evaluate both cases in Section 5.2, and we implement the no-parameters case.

**Preserved semantics.** The transformations must preserve the functionality and behavior of the original sample, e.g., the malicious behavior of Android malware. To check if the application's behavior has been kept unchanged after the injection, one could build a suite of automatic tests to perform basic operations. For instance, it is possible to open and close the main `activity`, put it in the background, then verify if the produced output is the same as the one of the original app. In our setting, the main criticality of the injection of API calls is related to the execution of operations that could lead to crashes or block the execution, which is especially relevant when calling methods, while more manageable when calling only class constructors. More specifically, a call may require a reference to non-existent objects, causing an exception in the execution (e.g., `openOptionsMenu()` from `android.view.View` if no Option Menu is present) or may block the user interface if it runs in the main thread.

**Plausibility.** The created adversarial samples have to be plausible for human inspection, i.e., they do not contain evident (from a human perspective) signs of manipulation. For example, having

50 consecutive calls of the same method inside the same function would be extremely suspicious. However, this concept is also quite tricky in practice; in fact, there are no general and automatic approaches to evaluate it. In our work, we pursue this constraint by limiting the repetition of the same calls multiple times in the same portion of the app. In particular, we spread the injected calls throughout the whole app in order to make a sequence of constructor calls less likely. However, a more sophisticated strategy should take care of the coherence of the injected code with the application context. For instance, adding permissions that do not pertain to the app's scope could be suspicious to the human expert.

**Robustness.** The alterations made to the samples should be resilient to preprocessing. For example, injecting dead code in the app is common for attackers, but it is easy to neutralize through dead code removal tools. In this sense, our approach aims at the injection of code that is properly executed.

#### 4.2.2. API Injection Feasibility

In the specific setting of this work, successfully creating the adversarial samples implies carefully selecting the system APIs to inject. Therefore, to implement the attack, the first step is to identify what API constructors are usable. Starting from the complete set of them for each package of each API level, we remove the ones that (a) are not public or have protected access, (b) belong to abstract classes, (c) potentially throw exceptions (thus requiring a more complex code injection), and (d) receive parameters of non-primitive types. Then, we identify as usable the classes that have at least one constructor satisfying this filtering; consequently, we also derive the packages that have at least one class available. Moreover, we consider two cases on the input parameters of the constructors: the first one—which we call no-parameters—computes the values on the basis of the constructors that receive no parameters in order to be called; in the second case—which we call primitive-parameters—we also include constructors that receive parameters of primitive types, where by primitive, we mean one among the following types: `int`, `short`, `long`, `float`, `double`, `char`, and `boolean`. Notably, attackers could include other non-primitive types that are simple to manage, such as `java.lang.String`. In Section 5.2, we evaluate the attained results quantitatively.

**Explaining evasion.** Besides identifying the modifiable packages and classes at the disposal of the attacker, it would be interesting to understand if and to what extent the usable APIs turn out to be the ones that are effective for evasion attacks; that is, the ones that, when modified, move the adversarial sample towards the benign class.

To perform this kind of evaluation, we make use of integrated gradients [35]. This is a state-of-the-art interpretability technique (for the sake of simplicity, in this paper, we also use the term explainability interchangeably), part of the so-called attribution techniques. As illustrated by Ancona et al. [36], this term means that the explanation of a sample $z$ consists of a vector $\boldsymbol{r}^z = [r_1, r_2, ..., r_d]$, where each component is a real value—an attribution or relevance—associated with each feature. This value can be positive or negative, depending on the direction in which the feature orients the classification. As regards to integrated gradients specifically, it is a gradient technique since it is based on the computation of the partial derivative of the prediction function $f$ with respect to each feature of the input sample vector $x_i$. Different from other similar techniques, it is designed to deal with non-linear classifiers. We refer the reader to the work by Ancona et al. [36] for further details.

The relevance values produced by this kind of technique are associated with each feature of a single sample (local technique) and do not provide an estimate of the general importance of the feature for the classifier (such as for global techniques). The relationship between interpretability and adversarial attacks is currently under study. For example, recent work has proposed to put a bridge between gradient-based explainability techniques (like integrated gradients) and evasion attacks specifically in the Android malware detection domain, showing the correlation between relevance values and the vulnerability of detectors to sparse evasion attacks [37]. In the following, we then assume that a relevant feature is significant for a successful realization of the evasion attack.

Since the attribution values of integrated gradients can be calculated with respect to a specific output class of the classifier, we consider the trusted one. In this way, positive values indicate that a feature moves the prediction towards the trusted class. Consequently, we consider a feature as relevant (for evasion) when its attribution value is strictly positive. In other words, we identify the features that influence the classification in the direction of the trusted class and, consequently, the ones that an attacker should modify to evade the detection of the considered sample. We show this assessment in Section 5.2.

## *4.3. Adversarial Malware Creation*

The core of our implementation is based on two libraries: `DexLib` [38] and `Apktool` [39]. `Dexlib` is a Java library with a great number of functionalities, such as reading, modifying, and writing Android `.dex` files. `Apktool` is a tool to disassemble, reassemble, and debug an Android application. Figure 3 shows the architecture of our implementation of the system to generate adversarial samples according to a mimicry attack—or a random noise one alternatively. The system takes as input a malicious Android sample and gives as output an adversarial malware that is expected to be classified as benign. The system chain consists of three phases that we describe in the following.

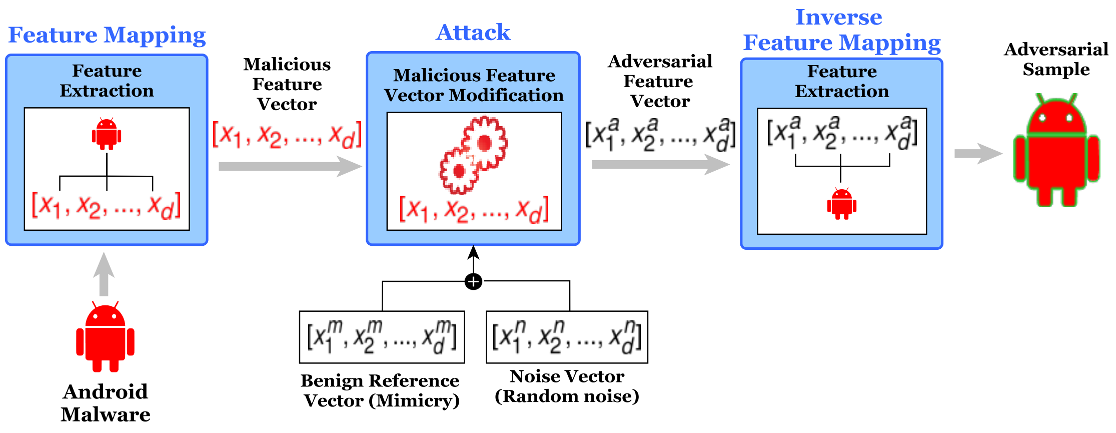

Feature Mapping
Feature Extraction
$[x_1, x_2, ..., x_d]$
Android Malware
Malicious Feature Vector
$[x_1, x_2, ..., x_d]$
Attack
Malicious Feature Vector Modification
$[x_1, x_2, ..., x_d]$
$[x_1^m, x_2^m, ..., x_d^m]$
Benign Reference Vector (Mimicry)
$[x_1^n, x_2^n, ..., x_d^n]$
Noise Vector (Random noise)
Adversarial Feature Vector
$[x_1^a, x_2^a, ..., x_d^a]$
Inverse Feature Mapping
Feature Extraction
$[x_1^a, x_2^a, ..., x_d^a]$
Adversarial Sample

**Figure 3.** The architecture of the system. Firstly, it processes the given malicious sample to retrieve its feature vector. Then, it performs the modifications to the feature vector using either the benign reference vector (mimicry) or the noise vector (random noise adding). Finally, it generates the adversarial malicious sample.

**Feature Mapping.** In this phase, the malicious Android sample is statically analyzed to detect and count all the Android system API calls and create the numeric vector of features $\boldsymbol{x} = [x_1, x_2, ..., x_d]$. As shown in Figure 4, firstly, we parse the `.dex` files (using `Dexlib`) to get the API called through the invoke functions (see Section 2), matching the Android system calls previously gathered from the official Android documentation. Then, we count the occurrences for each API call. Finally, we generate a sparse feature vector where, for every non-zero feature, there is the occurrence of the references to the specific package inside the analyzed Android application.

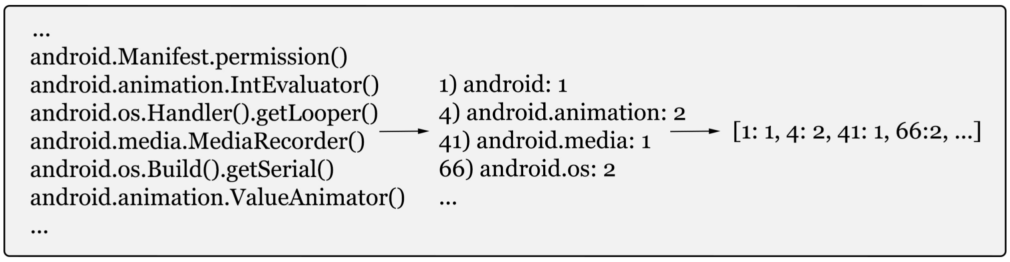

...
android.Manifest.permission()
android.animation.IntEvaluator()
android.os.Handler().getLooper()
android.media.MediaRecorder()
android.os.Build().getSerial()
android.animation.ValueAnimator()
...
1) android: 1
4) android.animation: 2
41) android.media: 1
66) android.os: 2
...
[1: 1, 4: 2, 41: 1, 66:2, ...]

**Figure 4.** Example of feature mapping for the creation of the feature vector.

**Attack.** The extracted malicious feature vector is then modified to perform the attack. This phase aims to generate an adversarial feature vector using a mimicry or a random noise addition approach. As shown in Figure 3, for the mimicry case, we choose as a reference a unique benign feature vector $\boldsymbol{x}^m = [x^m{}_1, x^m{}_2, ..., x^m{}_d]$. The choice of this vector can be made in different ways. Specifically, one might choose the benign reference vector to be added to the malicious sample according to different strategies. In Section 5.3.1, we compare the results of our experiments for four ways of choice, which we call: mean, median, real mean, and real median. Basically, in the first two cases, we take as the reference vector the mean (median) vector among the trusted samples available to the attacker, i.e., the test set; in the remaining two cases, we consider the real sample of the test set that is closest to the mean (median) vector. Specifically, we take the real sample with the highest cosine similarity, calculated with respect to the reference feature vector through the following formulation:

$$\text{Cosine Similarity}(\boldsymbol{x}, \boldsymbol{x}^m) := \frac{\sum_{i=1}^{d} x_i x^m{}_i}{\|\boldsymbol{x}\| \|\boldsymbol{x}^m\|} \tag{1}$$

As regards the random noise addition, we generate a noise vector $\boldsymbol{x}^n = [x^n{}_1, x^n{}_2, ..., x^n{}_d]$. We consider different implementation strategies for the attack (see Section 5.3.2) also in this case. We define these strategies as absolute and relative. The first one consists of increasing, for each feature, the occurrence of the correspondent call with a randomly chosen value between zero and the considered noise level (e.g., 10). In the second one, the features are added by taking into account the original value of each feature in the sample. For example, with an original feature value of 20 and a noise level of 50%, we randomly increase the occurrence with a value between zero and 10.

Once we obtain the vector that enables the modification, for the mimicry case, we compute the difference between the reference feature vector and each malicious one, then add the resulting features to the malicious sample, creating the adversarial feature vector $\boldsymbol{x}^a = [x^a{}_1, x^a{}_2, ..., x^a{}_d]$. Notably, if the original malicious sample exhibits one or more features with values higher than those of the reference vector, we keep the same value of the original sample (otherwise, we would have performed feature removal). Regarding the random noise addition case, the noise vector is added to the malicious feature vector to create the adversarial feature vector $\boldsymbol{x}^a$.

**Inverse Feature Mapping.** This phase is the opposite of the feature mapping phase, so each value of the adversarial feature vector, which is not already in the malicious sample, is matched with the corresponding Android system call and added in the malicious sample. We use `Apktool` to disassemble the Android application; then, we employ `Dexlib` to perform all the modifications on the bytecode level. Finally, we leverage `Apktool` again to reassemble and sign the generated adversarial sample, which is manually tested to verify that the functionality is preserved. As introduced in Section 4.2, for each feature (package), there may be more than one usable constructor since multiple classes can be available. Thus, for each feature, we randomly choose a constructor among the available ones. In this way, we increase the plausibility of the adversarial app, as it would be easier for a code analyst to notice the same class called multiple times rather than different classes of the same package. In this sense, we also spread the injected calls across all the methods already defined in the `.dex` file, so that they are not concentrated in a unique portion of code.

Each injected call is added by defining an object of the related class and calling its constructor. Figure 5 shows the Smali representation of the injection code. We can see a new-instance function to create an object of the class to add and an invoke-direct function to call the constructor of that class. The injected instructions are placed before the `return` statement of the selected method. Notably, the choice of calling constructors does not cause any side-effect since no features other than the target ones are modified within the feature vector.

```
new-instance v0, java.util.concurrent.atomic.AtomicLong;
invoke-direct v0, java.util.concurrent.atomic.AtomicLong;-><init>()V
```

**Figure 5.** Example of the injection of an Android system call.

## 5. Experimental Results

In this section, after detailing our experimental setup (Section 5.1), we evaluate the capability for the attacker to inject Android's system API calls (Section 5.2). Then, we show the performance of the mimicry and the random noise attacks (Section 5.3), as well as their impact in terms of added calls with respect to the original samples (Section 5.4).

### *5.1. Setting*

**Dataset.** We use the same dataset as [21], composed of 39,157 Android applications, 3017 of which are ransomware applications, 18,396 trusted applications, and 17,744 malicious applications (that do not include ransomware). As concerns the ransomware samples, they were gathered from the `VirusTotal` [40] service and the `HelDroid` dataset [41]. The malware samples were retrieved from (a) the `Drebin` dataset [10], (b) `Contagio`, a free source of malicious mobile applications, and (c) `VirusTotal`. As concerns the trusted applications, they came from the Google Play store, thanks to an open-source crawler [42], and from the Androzoo [43] dataset.

**Feature extraction.** As described in Section 2.2, we used R-PackDroid [12] as a reference detection system. The feature set consists of the cumulative list of system API packages up to Level 26 (Android Oreo), for a total of about 200 features. In particular, in our case, the set of APIs was strictly limited to the to the Android platform [44] ones.

**Classifier.** We trained an MLP (multilayer perceptron) neural network with Keras [45]. We performed a five-fold cross-validation repeated 100 times, along with a search for the best hyperparameters (e.g., number of layers, number of neurons per layer) for the net. We performed a random split for each repetition, with 50% of the dataset used as the training set. Figure 6 shows the mean ROC curve over the five repetitions; we performed all the experiments using the best classifier of the first iteration.

Notably, the detection performance was not the optimal one for this setting since, in our tests, we attained better results with a random forest algorithm. However, explanations with integrated gradients cannot be produced from random forest due to the non-differentiability of its decision function. Therefore, to keep our setting coherent, we chose to perform all the experiments on the MLP classifier.

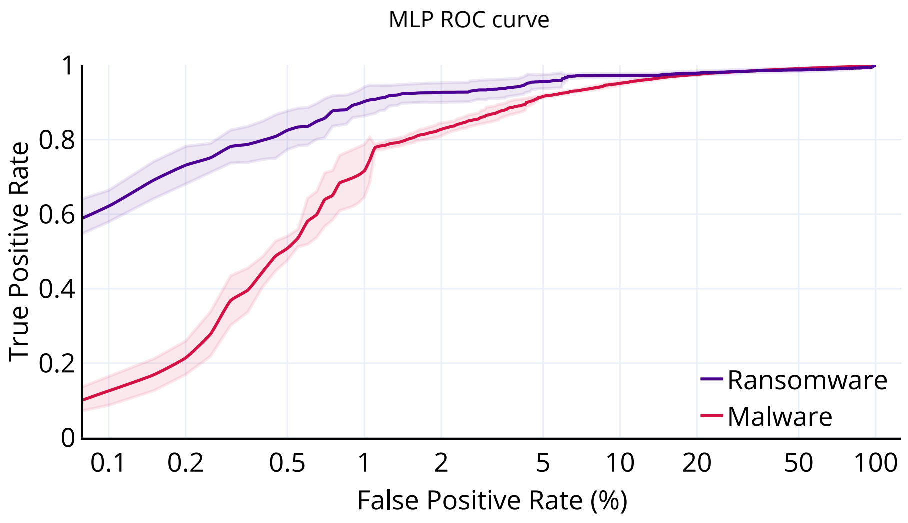


**Figure 6.** Average ROC curve of the MLP classifier over the five repetitions of the 10-fold cross-validation. The lines for the ransomware and malware classes include the standard deviation in translucent color.

### *5.2. API Injection Evaluation*

As explained in Section 4.2, attackers have to take into account the feasibility of the alterations of their malicious samples. Table 1 shows, for each API level up to 29, the percentage of usable packages and classes out of the whole set of APIs. In particular, Table 1a is related to the no-parameters case. Depending on the Android API level, it is possible to cover several packages, from 51% to 57%, and several classes between 15% and 27%.

In the second case—primitive-parameters—of Table 1b, we also include constructors that receive parameters of primitive types. With this variation, it is possible to cover more packages, between 55% and 61%, and more classes, between 18% and 33%, depending on the Android API level. Overall, we point out that the results are similar in both cases, but the effort in the first case is lower since that attacker does not need to create and inject new variables with the correct primitive type. Therefore, the first approach is more convenient for an attacker that wants to generate adversarial samples.

It is worth noting that the attacker's goal is to infect the highest number of devices possible. Consequently, the minimum API level of a malicious sample tends to be very low. For example, by extracting the `sdkversion` field in the `Manifest` of each ransomware sample of our test set, we verified that several apps lie in the 7–9 range. Therefore, attackers are encouraged to inject older APIs rather than the newer ones.

**Table 1.** Number of available packages and classes for each Android API level. In Table 1a, only constructors without parameters are considered, while in Table 1b, we also include constructors that receive primitive parameters.

| **Granularity (%)** | **Android API Level** | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| **Packages** | 53 | 54 | 55 | 56 | 56 | 56 | 55 | 55 | 55 | 56 | 57 | 57 | 56 | 56 | 55 | 55 | 55 | 55 | 55 | 54 | 53 | 53 | 53 | 53 | 51 | 51 | 51 | 51 |
| **Classes** | 27 | 26 | 26 | 25 | 25 | 25 | 24 | 24 | 24 | 24 | 24 | 24 | 23 | 23 | 22 | 22 | 21 | 21 | 21 | 19 | 19 | 18 | 18 | 18 | 17 | 17 | 16 | 15 |

**(a)**

| **Granularity (%)** | **Android API Level** | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| **Packages** | 58 | 59 | 59 | 61 | 61 | 61 | 60 | 59 | 59 | 60 | 61 | 61 | 60 | 60 | 59 | 59 | 58 | 58 | 58 | 57 | 58 | 58 | 58 | 58 | 56 | 56 | 55 | 55 |
| **Classes** | 33 | 32 | 32 | 31 | 31 | 31 | 30 | 29 | 29 | 29 | 29 | 29 | 28 | 28 | 27 | 26 | 26 | 25 | 25 | 23 | 23 | 22 | 21 | 21 | 20 | 20 | 19 | 18 |

**(b)**

**Are the modifiable features effective for evasion attacks?** The number of available packages and classes inferred in the previous experiment appears to be not really high. However, as introduced in Section 4.2.2, it is also worth inspecting the importance of the usable features to the classification. From now on, we conduct all of our experiments using the ransomware samples of the test set, which emulates the set of samples at the attacker's disposal. We computed the explanations using `DeepExplain` [46].

As a first point, we evaluate the percentage of relevant features modifiable by the attacker for each sample. We attain a mean value across the samples of 72.1% for the no-parameters case and of 73.8% in the primitive-parameters one. This result suggests that the attacker can modify a good number of useful features to evade detection. As a second test, we identify which relevant features are the most frequent among the modifiable ones. The results are shown in Figure 7. As can be seen, the shown features correspond, as expected, to the ones that are known to be descriptive of the trusted samples, i.e., a broad set of functionalities related, for example, to the app's user interface.

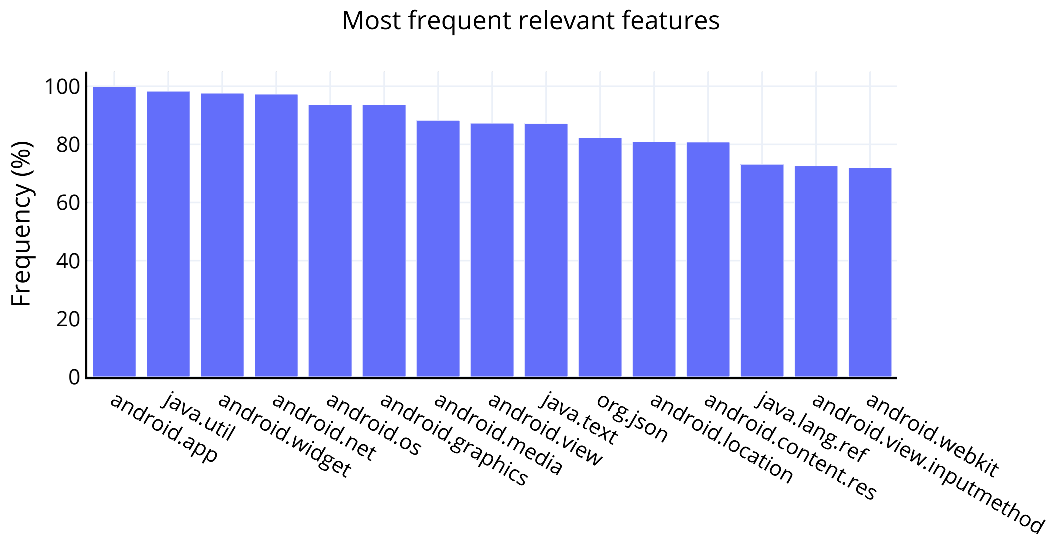


**Figure 7.** Top 15 relevant features among the usable ones.

## 5.3. Attack Results

In the following, we assess the mimicry attack's performance and, as a comparison, the most straightforward attack possible, the random noise one.

### 5.3.1. Mimicry Attack

Figure 8a shows the evasion rate for four different reference samples, as illustrated in Section 4.3; that is, how many adversarial samples evade the classification. In the $x$-axis, we evaluate an increasing percentage of modified features, i.e., the number of injected packages out of the total number of modifiable ones by the attacker. Since the choice of the subset of features to modify for each percentage level below 100% is random, we show the average result over five repetitions. Moreover, it is worth noting that each chosen feature exhibits a different number of calls to inject because some APIs are being called in the app thousands of times, while others might be referenced only a few times. In the same figure, it is possible to see that all the curves present similar trends.

In Figure 8b, we focus on the median strategy and show how the samples are classified. We can see that the evasion works as desired because the number of samples classified as trusted increases while the number of samples classified as ransomware and malware decreases. Notably, in Figure 8a, the evasion rate at 0% of modified features is not zero because some ransomware samples are mistakenly classified as benign or malware. For the same reason, Figure 8b shows that the detection of the samples as ransomware is not 100%.

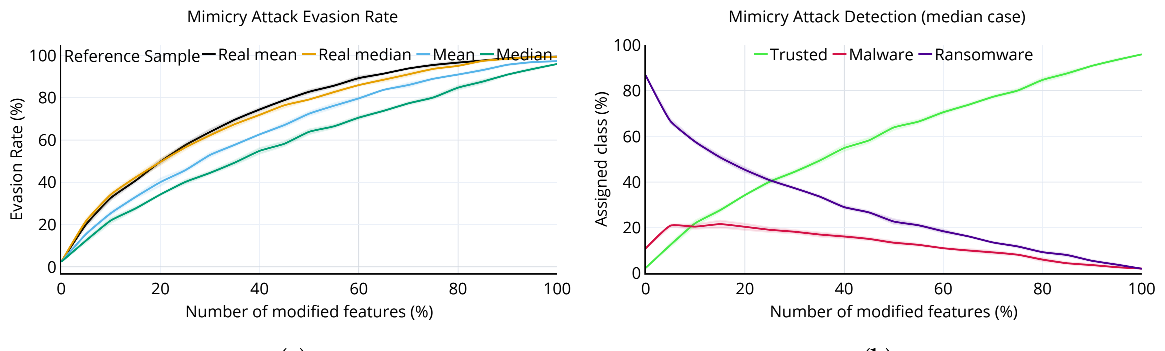


**Figure 8.** Mimicry attack's evasion rate for different choices of the reference sample (**a**). Detection rate detail of the ransomware samples (**b**). Both figures are the average results over five repetitions and include the standard deviation in translucent color.

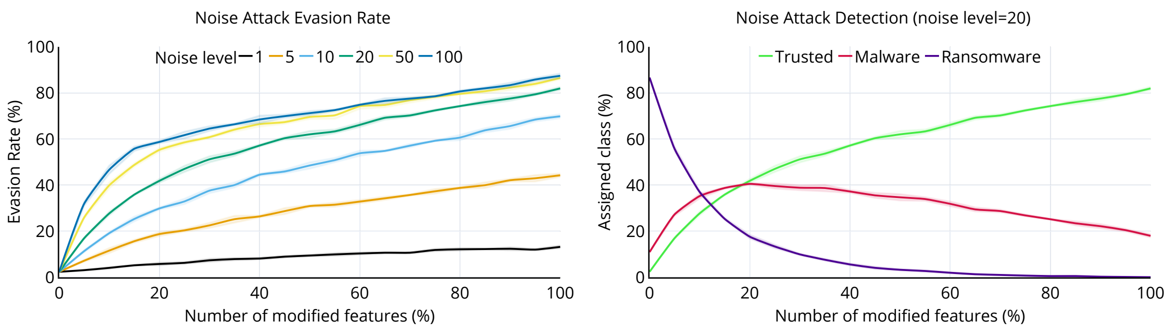
#### 5.3.2. Random Noise Attack

After the mimicry attack, we performed a random noise addition attack, which can be useful as a reference since it only consists of injecting API calls without any specific pattern or target class. Following the same procedure as Section 5.3.1, the results are shown in Figure 9. Specifically, the absolute approach is the one shown in Figure 9a. On the left side, we evaluate the evasion rate for different levels of added noise. As we can see, the higher the noise level is, the higher the evasion rate is. On the right side of the figure, we show the detail of the assigned classes for a noise level equal to 20, which achieves similar evasion levels as the mimicry case. As we can see, the curve also exhibits the same tendency, i.e., it causes an increasing detection of the ransomware samples as legitimate ones as if it was a targeted attack. This is significant since it seems to suggest that no specific injection pattern is needed to make ransomware samples classified as trusted ones. Consequently, attackers would not need a set of trusted samples with the same probability distribution of the target system's training set, which is necessary to perform the mimicry attack.

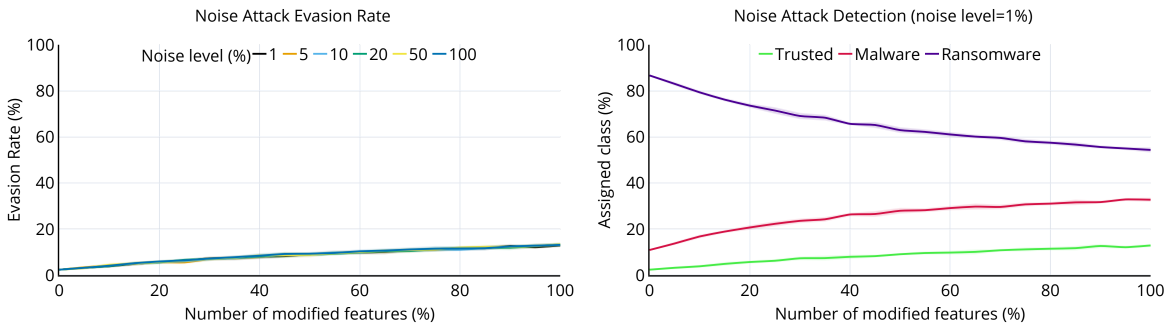
The relative approach is shown in Figure 9b. The evasion rate on the left side of the figure is, in this case, completely different from the absolute one, reaching a value of around 15% at the highest point. Notably, there is no significant difference between each noise level. This can be related to the sparsity of the samples' feature vector. In fact, several features have a zero value, so the percentage of a zero value would always end up with no addition. Therefore, in our implementation, we chose to set a random increase between zero and one. Ultimately, this result shows that adding a high percentage of noise only to the features that already had non-zero values is insufficient to accomplish the evasion.

### *5.4. Injection Impact*

The mimicry and the random noise addition attack results showed that the evasion rates can go up to around 80%. This appears to be an outstanding result; however, it does not depict the full view of the problem. For example, it does not tell anything about the plausibility of the adversarial sample. In fact, we may say that the plausibility for the method proposed in this paper is inversely correlated to the number of injected features. The more additional calls (with respect to the original sample) there are, the lower is the plausibility. Therefore, with Figure 10, we evaluate the impact on the samples of both the considered attacks (median case for the mimicry attack, absolute noise level equal to 20 for the noise attack), in terms of extra calls added. As with the previous Figures, in the $x$-axis, we put an increasing percentage of modified features. We insert two $y$-axes to show the average number of added calls both as a percentage of the original amount (left axis) and as the absolute number (right axis). As can be seen in Figure 10a, the mimicry attack causes a massive increase in calls. For example, let us fix the evasion rate to 50%, which requires modifying around 35% of the modifiable features (see Figure 8a). With this setting, the increment of system API calls would be almost nine thousand percent on average. Notably, the standard deviation is quite high, so this number can be significantly higher or lower; nevertheless, the generated adversarial sample would most likely not be realistic.



(**a**) Absolute noise addition



(**b**) Relative noise addition

**Figure 9.** Evasion rate for different levels of noise injection in the absolute and relative cases (left side of Figure 9a,b, respectively). Detection detail for a noise level equal to 20 (right side of Figure 9a) and a 1% noise level (right side of Figure 9b). All the results are the average over five repetitions and include the standard deviation in translucent color.

The random noise addition attack attains much better results in this sense. A 50% evasion rate is accomplished by modifying around 30% of the modifiable features (see Figure 9a), which means injecting 69% additional calls with respect to the original samples on average. Although the difference with the mimicry attack is significant, the level of additional calls to add results in being too high to be plausible.

Overall, the results showed the detector's vulnerability to perturbed inputs, even with a non-targeted attack. However, to achieve a sufficient evasion rate, attackers need to inject a vast number of calls, which weakens the attack's plausibility.

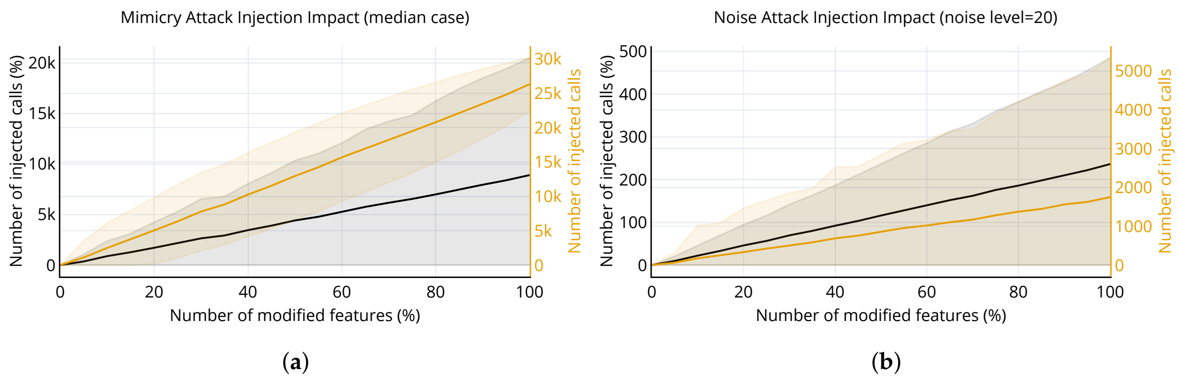


**Figure 10.** Average impact on the number of calls for mimicry (**a**) and noise (**b**) attacks. The standard deviation is also reported in translucent color.

## 6. Conclusions, Limitations, and Future Work

In this work, we present a study about the feasibility of performing a fine-grained injection of system API calls to Android apps in order to evade machine learning-based malware detectors. This kind of strategy can be particularly effective for creating a massive amount of adversarial samples with a relatively low effort by the attackers. However, we discuss the necessity of satisfying several constraints to generate realistic, working samples. In fact, the experimental results show that both the mimicry and the random noise attacks, which do not require a high level of knowledge about the target system, suffice to evade classification. Nevertheless, they cause a substantial loss of plausibility of the generated adversarial sample since they add an excessive number of additional calls, thus weakening the effectiveness. Therefore, we believe that future work should focus on implementing an evasion attack using a gradient approach, minimizing the number of injected features to preserve the plausibility. This aspect is also relevant to highlight that the detector considered in our work employs non-binary features (different from all previous articles in the literature). Consequently, we think an interesting future work would be to compare the impact of the feature vector design on the practical feasibility of adversarial malware creation.

We also point out that our specific implementation to address the inverse feature-mapping problem is not optimal as well in terms of plausibility. In fact, other work in the literature—such as the one by Pierazzi et al. [19] where portions of real code were injected through automated software transplantation—present, among others, the potential issue of injecting code that is out of context with respect to the original app. In our case, the main concern lies instead in the choice of injecting calls to class constructors. That is, we call new objects that are not used in the rest of the code. Therefore, a more plausible solution for future work could be the injection of method calls, which are more likely to be realistic even without being referenced next in the code.

**Author Contributions:** Data curation, M.S.; Funding acquisition, G.G. and D.M.; Methodology, F.C. and M.S.; Software, F.C. and M.S.; Supervision, G.G. and D.M.; Validation, G.G. and D.M.; Writing – original draft, F.C.; Writing – review and editing, F.C., M.S. and D.M. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was partly supported by the project PON AIM Research and Innovation 2014-2020—Attraction and International Mobility, funded by the Italian Ministry of Education, University and Research.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. McAfee. *McAfee Mobile Threat Report*; McAfee: Santa Clara, CA, USA, 2020.
2. Kaspersky. IT Threat Evolution Q3 2019. Statistics. Available online: https://securelist.com/it-threat-evolution-q3-2019-statistics/95269 (accessed on 29 October 2019).

3. Feng, Y.; Anand, S.; Dillig, I.; Aiken, A. Apposcopy: Semantics-based detection of Android malware through static analysis. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–22 November 2014; pp. 576–587. [CrossRef]
4. Arzt, S.; Rasthofer, S.; Fritz, C.; Bodden, E.; Bartel, A.; Klein, J.; Le Traon, Y.; Octeau, D.; McDaniel, P. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation—PLDI'14, Edinburgh, UK, 9–11 June 2014; ACM Press: New York, NY, USA, 2013; pp. 259–269. [CrossRef]
5. Zhou, Y.; Wang, Z.; Zhou, W.; Jiang, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. *NDSS* **2012**, *25*, 50–52.
6. Tam, K.; Khan, S.J.; Fattori, A.; Cavallaro, L. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In Proceedings of the 2015 Network and Distributed System Security Symposium, Internet Society, San Diego, CA, USA, 8–11 February 2015; doi:10.14722/ndss.2015.23145. [CrossRef]
7. Zhang, Y.; Yang, M.; Xu, B.; Yang, Z.; Gu, G.; Ning, P.; Wang, X.S.; Zang, B. Vetting undesirable behaviors in android apps with permission use analysis. In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security—CCS '13, Berlin, Germany, 4–8 November 2013; ACM Press: New York, NY, USA, 2013; pp. 611–622. [CrossRef]
8. Chen, J.; Wang, C.; Zhao, Z.; Chen, K.; Du, R.; Ahn, G.J. Uncovering the Face of Android Ransomware: Characterization and Real-Time Detection. *IEEE Trans. Inf. Forensics Secur.* **2018**, *13*, 1286–1300. [CrossRef]
9. Fereidooni, H.; Conti, M.; Yao, D.; Sperduti, A. ANASTASIA: ANdroid mAlware detection using STatic analySIs of Applications. In Proceedings of the 2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS), Larnaca, Cyprus, 21–23 November 2016; doi:10.1109/NTMS.2016.7792435. [CrossRef]
10. Arp, D.; Spreitzenbarth, M.; Hübner, M.; Gascon, H.; Rieck, K. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In Proceedings of the 2014 Network and Distributed System Security Symposium. Internet Society, San Diego, CA, USA, 23–26 February 2014; doi:10.14722/ndss.2014.23247. [CrossRef]
11. 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—ASIA CCS'16, Xi'an, China, 30 May–3 June 2016; ACM Press: New York, NY, USA, 2016; pp. 377–388. [CrossRef]
12. Maiorca, D.; Mercaldo, F.; Giacinto, G.; Visaggio, C.A.; Martinelli, F. R-PackDroid: API package-based characterization and detection of mobile ransomware. In Proceedings of the Symposium on Applied Computing—SAC'17, Santa Fe, NM, USA, 13–17 March 2005; ACM Press: New York, NY, USA, 2017; pp. 1718–1723. [CrossRef]
13. Yuan, Z.; Lu, Y.; Wang, Z.; Xue, Y. Droid-Sec: Deep learning in android malware detection. In Proceedings of the 2014 ACM Conference on SIGCOMM—SIGCOMM'14, Chicago, IL, USA, 17–22 August 2014; ACM Press: New York, NY, USA, 2014; pp. 371–372. [CrossRef]
14. Demontis, A.; Melis, M.; Biggio, B.; Maiorca, D.; Arp, D.; Rieck, K.; Corona, I.; Giacinto, G.; Roli, F. Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection. *IEEE Trans. Dependable Secur. Comput.* **2019**, *16*, 711–724. [CrossRef]
15. Biggio, B.; Corona, I.; Maiorca, D.; Nelson, B.; Šrndić, N.; Laskov, P.; Giacinto, G.; Roli, F. Evasion Attacks against Machine Learning at Test Time. In *Advanced Information Systems Engineering*; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7908, pp. 387–402._25. [CrossRef]
16. Maiorca, D.; Corona, I.; Giacinto, G. Looking at the bag is not enough to find the bomb: An evasion of structural methods for malicious pdf files detection. In Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, Hangzhou, China, 8–10 May 2013; pp. 119–130.
17. Melis, M.; Maiorca, D.; Biggio, B.; Giacinto, G.; Roli, F. Explaining black-box android malware detection. In Proceedings of the IEEE 2018 26th European Signal Processing Conference (EUSIPCO), Rome, Italy, 3–7 September 2018; pp. 524–528.
18. Quiring, E.; Maier, A.; Rieck, K. Misleading Authorship Attribution of Source Code using Adversarial Learning | USENIX. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, USA, 14–16 August 2019; pp. 479–496.

19. Pierazzi, F.; Pendlebury, F.; Cortellazzi, J.; Cavallaro, L. Intriguing Properties of Adversarial ML Attacks in the Problem Space. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020, pp. 1332–1349. [CrossRef]
20. Song, W.; Li, X.; Afroz, S.; Garg, D.; Kuznetsov, D.; Yin, H. Automatic Generation of Adversarial Examples for Interpreting Malware Classifiers. *arXiv* **2020**, arXiv:2003.03100.
21. Scalas, M.; Maiorca, D.; Mercaldo, F.; Visaggio, C.A.; Martinelli, F.; Giacinto, G. On the effectiveness of system API-related information for Android ransomware detection. *Comput. Secur.* **2019**, *86*, 168–182. [CrossRef]
22. Maiorca, D.; Ariu, D.; Corona, I.; Aresu, M.; Giacinto, G. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Comput. Secur.* **2015**, *51*, 16–31. [CrossRef]
23. Faruki, P.; Bharmal, A.; Laxmi, V.; Ganmoor, V.; Gaur, M.S.; Conti, M.; Rajarajan, M. Android Security: A Survey of Issues, Malware Penetration, and Defenses. *IEEE Commun. Surv. Tutor.* **2015**, *17*, 998–1022. [CrossRef]
24. Zhang, F.; Huang, H.; Zhu, S.; Wu, D.; Liu, P. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks, Oxford, UK, 23–25 July 2014; pp. 25–36.
25. Barreno, M.; Nelson, B.; Joseph, A.D.; Tygar, J.D. The security of machine learning. *Mach. Learn.* **2010**, *81*, 121–148. [CrossRef]
26. Barreno, M.; Nelson, B.; Sears, R.; Joseph, A.D.; Tygar, J.D. Can machine learning be secure? In Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security—ASIACCS '06, Taipei, Taiwan, 21–24 March 2006; p. 16. [CrossRef]
27. Dalla Preda, M.; Madou, M.; De Bosschere, K.; Giacobazzi, R. Opaque Predicates Detection by Abstract Interpretation. In *Algebraic Methodology and Software Technology*; Springer: Berlin/Heidelberg, Germany, 2006; Volume 4019, pp. 81–95._9. [CrossRef]
28. Ming, J.; Xu, D.; Wang, L.; Wu, D. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security—CCS'15, Denver, CO, USA, 12–16 October 2015; pp. 757–768. [CrossRef]
29. Grosse, K.; Papernot, N.; Manoharan, P.; Backes, M.; McDaniel, P. Adversarial Examples for Malware Detection. In Proceedings of the Computer Security—ESORICS 2017, Oslo, Norway, 11–15 September 2017; pp. 62–79.
30. Yang, W.; Kong, D.; Xie, T.; Gunter, C.A. Malware Detection in Adversarial Settings: Exploiting Feature Evolutions and Confusions in Android Apps. In Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, 4–8 December 2017; pp. 288–302. [CrossRef]
31. Yang, W.; Xiao, X.; Andow, B.; Li, S.; Xie, T.; Enck, W. AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context. In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; Volume 1, pp. 303–313. [CrossRef]
32. Rosenberg, I.; Shabtai, A.; Rokach, L.; Elovici, Y. Generic Black-Box End-to-End Attack Against State of the Art API Call Based Malware Classifiers. *arXiv* **2018**, arXiv:1707.05970.
33. Hu, W.; Tan, Y. Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN. *arXiv* **2017**, arXiv:1702.05983.
34. Li, J.; Madry, A.; Peebles, J.; Schmidt, L. Towards Understanding the Dynamics of Generative Adversarial Networks. *arXiv* **2017**, arXiv:1706.09884.
35. Sundararajan, M.; Taly, A.; Yan, Q. Axiomatic Attribution for Deep Networks. In Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, 6–11 August 2017; pp. 3319–3328.
36. Ancona, M.; Ceolini, E.; Öztireli, C.; Gross, M. Gradient-Based Attribution Methods. In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*; Springer: Cham, Switzerlands, 2019; Volume 11700, pp. 169–191._9. [CrossRef]
37. Melis, M.; Scalas, M.; Demontis, A.; Maiorca, D.; Biggio, B.; Giacinto, G.; Roli, F. Do Gradient-based Explanations Tell Anything About Adversarial Robustness to Android Malware? *arXiv* **2020**, arXiv:2005.01452.
38. Smali/Baksmali. Available online: https://github.com/JesusFreke/smali (accessed on 29 October 2019).
39. Apktool. Available online: https://ibotpeaches.github.io/Apktool (accessed on 29 October 2019).
40. VirusTotal. Available online: https://www.virustotal.com (accessed on 29 October 2019).

41. Andronio, N.; Zanero, S.; Maggi, F. HelDroid: Dissecting and Detecting Mobile Ransomware. In *Research in Attacks, Intrusions, and Defenses*; Springer International Publishing: Cham, Switzerlands, 2015; pp. 382–404.
42. Python Market Android Library. Available online: https://github.com/liato/android-market-API-py (accessed on 29 October 2019).
43. 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 Workshop on Mining Software Repositories—MSR '16, Austin, TX, USA, 14–15 May 2016; pp. 468–471. [CrossRef]
44. Android API Reference. Available online: https://developer.android.com/reference/packages (accessed on 29 October 2019).
45. Keras. Available online: https://keras.io (accessed on 29 October 2019).
46. DeepExplain. Available online: https://github.com/marcoancona/DeepExplain (accessed on 29 October 2019).