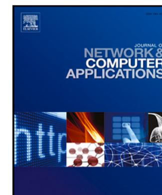
Journal of Network and Computer Applications 235 (2025) 104094

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca



Research paper

Light up that Droid! On the effectiveness of static analysis features against app obfuscation for Android malware detection

Borja Molina-Coronado [a,*], Antonio Ruggia [b], Usue Mori [c], Alessio Merlo [d], Alexander Mendiburu [a], Jose Miguel-Alonso [a]

[a] Dept. of Computer Architecture and Technology, University of the Basque Country UPV/EHU, Donostia, Spain
[b] Dept. of Informatics, Bioengineering, Robotics and Systems Engineering, University of Genoa, Genoa, Italy
[c] Dept. of Computer Science and Artificial Intelligence, University of the Basque Country UPV/EHU, Donostia, Spain
[d] CASD - Centre for Advanced Defense Studies, Rome, Italy

ARTICLE INFO

Keywords:
Machine learning
Static analysis
Malware detection
Obfuscation
Reliability
Evasion

ABSTRACT

Malware authors have seen obfuscation as the mean to bypass malware detectors based on static analysis features. For Android, several studies have confirmed that many anti-malware products are easily evaded with simple program transformations. As opposed to these works, ML detection proposals for Android leveraging static analysis features have also been proposed as obfuscation-resilient. Therefore, it needs to be determined to what extent the use of a specific obfuscation strategy or tool poses a risk for the validity of ML Android malware detectors based on static analysis features. To shed some light in this regard, in this article we assess the impact of specific obfuscation techniques on common features extracted using static analysis and determine whether the changes are significant enough to undermine the effectiveness of ML malware detectors that rely on these features. The experimental results suggest that obfuscation techniques affect all static analysis features to varying degrees across different tools. However, certain features retain their validity for ML malware detection even in the presence of obfuscation. Based on these findings, we propose a ML malware detector for Android that is robust against obfuscation and outperforms current state-of-the-art detectors.

1. Introduction

With the spread of Android devices, the amount of malware crafted for this OS has also experienced a extraordinary growth (Statista, 2021; Kaspersky Labs, 2021). This has led researchers to devise cutting-edge anti-malware solutions based on machine learning (ML) algorithms. When fed with app data, these algorithms are able to find patterns that are characteristic and informative enough to classify apps as either goodware or malware. In this sense, the performance of ML highly depends on the quality and soundness of the data that is used to build the classifier (Hastie et al., 2009; Molina-Coronado et al., 2020). In the case of Android malware detection, the extraction of this data, in the form of a vector of features that represents the behavior of apps, is performed using either dynamic or static analysis (Sadeghi et al., 2016; Wang et al., 2019).

Dynamic analysis is performed on a controlled environment (sandbox) where the app is executed. During execution, traces that describe the behavior of the app, e.g., network activity, system calls, etc. are logged (Sadeghi et al., 2016). On the contrary, static analysis is based on the inspection of the content of the package file (APK) of an app. This includes the compiled code and other resources such as image and database files (Li et al., 2017). Both techniques are valid to extract valuable data from apps. However, dynamic analysis involves a costly process whose success is dependent on the emulation method used and the absence of sandbox evasion artifacts in the code of apps. Instead, static analysis is computationally cheaper, but it can be counteracted by applying app code transformations. Such transformations are commonly known as obfuscation (Tam et al., 2017).

Obfuscation is a security through obscurity technique that aims to prevent automatic or manual code analysis. It involves the transformation of the code of apps, making it more difficult to understand but without altering its functionality (Sihag et al., 2021). This characteristic has made obfuscation a double edged sword, used by both, goodware and malware authors. Developers of legitimate software leverage obfuscation to protect their code from being statically analyzed by third parties, e.g., trying to avoid app repackaging or intellectual property abuses (Collberg and Thomborson, 2002). Malware authors have seen

* Corresponding author.
E-mail addresses: borja.molina@ehu.eus (B. Molina-Coronado), antonio.ruggia@dibris.unige.it (A. Ruggia), usue.mori@ehu.eus (U. Mori), alessio.merlo@ssuos.difesa.it (A. Merlo), alexander.mendiburu@ehu.eus (A. Mendiburu), j.miguel@ehu.eus (J. Miguel-Alonso).

https://doi.org/10.1016/j.jnca.2024.104094
Received 27 March 2024; Received in revised form 14 November 2024; Accepted 9 December 2024
Available online 19 December 2024
1084-8045/© 2025 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY-NC license (http://creativecommons.org/licenses/by-nc/4.0/).

obfuscation as a mean to conceal the purpose of their code (Dong et al., 2018), preventing static analysis tools from obtaining meaningful information about the behavior of apps.

It may seem common sense that the application of any, or the combination of several, obfuscation techniques will make malware analysis relying on features extracted using static analysis fruitless. However, it is unclear to what extent this aspect is true. Some studies on Windows and Android executables have demonstrated that obfuscation harms detectors that rely on static analysis features. For example, packing[1] prevents obtaining informative features (Aghakhani et al., 2020; Ruggia et al., 2021), which are essential to train malware detectors. Similar conclusions have been drawn for other forms of transformation (Hammad et al., 2018; Molina-Coronado et al., 2023; Gao et al., 2024), showing a major weakness in Android malware detectors. However, other studies contradicted these findings and proposed feature extraction techniques via static analysis that enable the successful identification of malware even when apps are obfuscated (Suarez-Tangil et al., 2017; Bacci et al., 2018; Garcia et al., 2018; Gao et al., 2023; Wu et al., 2022).

All these works appear promising in demonstrating either the flaws or the strengths of static analysis features for malware detection. However, their discrepancies complicate the extraction of sound conclusions regarding the validity of static analysis features for Android malware detection. Some limitations apply to these existing references. First, most studies based their analyses considering detectors as black boxes, without analyzing the impact of obfuscation on the apps and/or features used to train/evaluate them (Rastogi et al., 2014; Maiorca et al., 2015; Hammad et al., 2018; Bacci et al., 2018; Molina-Coronado et al., 2023; Gao et al., 2024). Nonetheless, this additional feature-centered analysis is important to understand and explain the reasons behind excellent or poor performance metrics when obfuscation is present, and is crucial for building more robust detectors. Additionally, a common flaw in many studies lies in the omission of details concerning their datasets and the configuration of their experimental setups (Suarez-Tangil et al., 2017; Garcia et al., 2018; Lee et al., 2019; Wu and Kanai, 2021). This absence of detail undermines reproducibility and leads to inconsistent findings between articles.

To shed light on the impact of obfuscation on static analysis features and the detectors that rely on them, this work presents the most comprehensive analysis to date on how common obfuscation techniques affect the information obtained through static analysis for Android malware detection using ML algorithms. The contributions of this paper can be summarized as follows:

- We provide an agnostic[2] evaluation of the strength, validity and detection potential of a complete set of features obtained by means of static analysis of APKs when obfuscation is used.
- We analyze the impact of a variety of obfuscation strategies and tools on static analysis features, providing insights about the use of these features for malware detection in obfuscated scenarios.
- We propose a high-performing ML-based Android malware detector leveraging a set of robust static analysis features. We demonstrate the ability of this detector to identify goodware and malware despite obfuscation, outperforming the state-of-the-art.
- We present a novel dataset with more than 95K obfuscated Android apps, allowing researchers to test the robustness of their malware detection proposals.
- In spirit of open science and to allow reproducibility, we make the code publicly available at https://gitlab.com/serralba/robustml_maldet.

The rest of this paper is organized as follows. Section 2 introduces the literature that has previously tackled obfuscation as a problem in malware analysis. Section 3 provided basic information about topics that are required to understand the content of this paper. Section 4 describes the construction of the app dataset and presents the features that are considered in our experiments. Section 5 evaluates the impact of different obfuscation strategies and tools in static analysis features, as well as their validity for malware detection. Section 6 is devoted to assess the robustness of our ML malware detection proposal. Section 7 includes a discussion of the main findings made along this paper. Finally, we conclude this paper in Section 8.

2. Related work

The related work can be divided into two groups: (1) studies that analyze the vulnerabilities of malware detectors when obfuscation is present, and (2) works that propose novel malware detectors which are presumably robust to obfuscation.

2.1. Study of the vulnerabilities of malware detectors

The works that evaluate the negative effects of obfuscation on Android malware detectors have mainly been carried out for black box malware detectors, i.e., the system or model is analyzed and evaluated based solely on its input–output behavior, without direct access to or knowledge of its internal workings. The first work of this type (Rastogi et al., 2014) studied how obfuscation impacts the detection ability of 10 popular anti-virus programs available in the VirusTotal platform. The work demonstrated that these detectors are vulnerable and loose their reliability in the identification of obfuscated malware. Similarly, in Maiorca et al. (2015), 13 Android anti-virus programs from VirusTotal are assessed using different obfuscation strategies to modify malware. The results showed a meek improvement in detection accuracy concerning the findings of previous works (Rastogi et al., 2014) and proved the effort of companies responsible of developing these tools to try to counteract obfuscation. A more comprehensive analysis for 60 anti-virus tools in VirusTotal has been presented in Hammad et al. (2018). Again, the work demonstrated the vulnerabilities of most detectors when facing obfuscated malware.

The mentioned studies showed that the success on bypassing detection highly depends on the obfuscation tools and strategies considered. They focused on commercial anti-virus tools, however, some other works have focused on assessing the impact of obfuscation in published ML based detectors. In Bacci et al. (2018), an analysis of the effect of obfuscation in two detectors, one relying on static and the other on dynamic analysis features, is presented. It is shown that the performance of the detector using dynamic analysis features is not altered by obfuscation, contrary to the detector that uses static analysis features. However, authors indicated that this effect can be easily mitigated by including obfuscated samples during the training phase of ML models. In Molina-Coronado et al. (2023), eight state-of-the-art Android malware detectors leveraging static analysis features and ML algorithms are assessed using obfuscated malware samples. The authors demonstrated that obfuscation is a major weakness of these popular solutions, since all of them suffered a drop in their performance. One of the most recent and comprehensive studies is carried out in Aghakhani et al. (2020). This work analyzes the effect of packing in ML malware detectors relying on static analysis for Windows executables. The conclusions drawn from the extensive set of experiments indicate that ML malware detectors for Windows fail to identify the class of transformed samples due to the insufficient informative capacity of static analysis features.

While all these studies highlighted the additional challenges that obfuscation introduces for malware detection, most of them fall short

[1] Packing is a particular form of obfuscation which hides the real code through one or more layers of compression/encryption. At runtime, the unpacking routine restores the original code in memory to be then executed.

[2] In this context, we refer agnostic as an analysis carried out without focusing on a specific malware detection proposal, but on elemental features extracted from app information. This allows anyone to infer how obfuscation can impact more complex features, that rely on this information, used by specific detection proposals.

in explaining the reasons behind accurate or erroneous detections when obfuscation is considered. This is due to the fact they consider the detectors as black-box tools to study the effect of obfuscation on their effectiveness. Since they do not analyze the effect of obfuscation on the apps, these works do not allow for a detailed analysis of how different obfuscation strategies and tools impact the information and features that can be used for detection. As an exception, in Gao et al. (2024), the authors examine the impact of obfuscation on the accuracy of 13 machine learning-based detectors. For analysis, the detectors are categorized into three groups: String-based, Image-based, and Graph-based. However, the broad categorization used for the study limits the ability to explain how obfuscation strategies impacts the information used detectors, specially for those using features from different file sources. Additionally, only a single obfuscation tool is considered, which may lead to inaccurate conclusions, as different obfuscation tools may implement the same obfuscation strategy in various ways. In contrast to these works, the goal of this paper is to provide comprehensive and detailed insights into the effect of obfuscation on Android apps and the different static analysis information, arranged into a fine-grained set of feature families, used for malware detection. This approach aims to contribute to the development of more effective detectors and to clarify the benefits of using static analysis features for detecting malware in different obfuscation scenarios.

2.2. Obfuscation-resilient detectors

A second group of studies focuses on the development of obfuscation-resilient detectors, specifically designed to operate effectively in the presence of obfuscated apps. AndroDet (Mirzaei et al., 2019) employs a online classifier that is incrementally retrained as new obfuscated samples are received, outperforming classical offline approaches. However, several experimental flaws have been discovered related to the lack of diversity on the obfuscated set used for training (Mohammadinodooshan et al., 2019). DroidSieve (Suarez-Tangil et al., 2017) categorizes static analysis features as obfuscation-sensitive and obfuscation-insensitive based on theoretical aspects. Feature frequency is studied for different datasets with obfuscated and unobfuscated malware samples to support the idea that most changing features provide better information. In consequence, they proposed a detector that relies on the features of both groups, and offering good performance in terms of malware detection and family identification. RevealDroid (Garcia et al., 2018) argues against static analysis features such as Permissions, Intents or Strings for robust malware detection. Contrary to the authors of DroidSieve, they suggest that obfuscation-sensitive features do not provide useful information to detect malware. Instead, the authors propose a new set of static analysis features based on a backward analysis of the calls to dynamic code loading and reflection APIs. In this way, the functions invoked at runtime are identified, nullifying the effect of obfuscation, making the proposed detector obfuscation-resilient. Recently, CorDroid (Gao et al., 2023) proposes the combination of the label of two models, one trained with opcode transition probabilities and other with sensitive API call graph information. The idea is to combine the output of the two models using and optimization procedure that weights them in order to obtain the final prediction.

Other allegedly obfuscation-resilient detectors leveraging deep learning algorithms are presented in Kim et al. (2018), Lee et al. (2019), Wu and Kanai (2021), Wu et al. (2022). The authors of these works suggest that the capacity of deep learning to embed and extract useful information from the features is enough to tackle obfuscation. SeqDroid (Lee et al., 2019) relies on strings extracted from the app code. Strings are then transformed into sequences of characters to obtain an embedding representation of the app that is then used for classification. Despite the excellent results reported for malware detection, the ability of the detector to identify obfuscated apps is based on (unproven) statements that are not specifically tested. In Wu and Kanai (2021) obfuscation-sensitive and insensitive features, including permissions, opcodes and meta-data from ApkID,[3] a signature-based fingerprinting tool are incorporated. Similarly to Lee et al. (2019), the obfuscation-resiliency of this work cannot be confirmed based on the results, since the effect of the use of obfuscation in the detector is based on theoretical aspects not specifically covered by the experiments. In Wu et al. (2022), a contrastive learning approach based on centrality metrics of API calls is used to extract features that distinguish between samples of different malware families even in obfuscation scenarios. Again, the robustness of the proposed detector cannot be ensured since the experimental setup does not include obfuscation techniques such as reflection, nor obfuscated goodware.

The experiments carried out in all these works present some flaws that, in our opinion, put in question their capability. For example, most of them do not describe, or vaguely analyze, the composition of their datasets in terms of the number of obfuscated malware or goodware samples or the obfuscation techniques applied to them (Kim et al., 2018). Also, the tools considered to obfuscate the samples in most cases are limited to only one, which can lead to implementation-specific biases during evaluation (Wu et al., 2022; Gao et al., 2023). Some articles focus their analyses exclusively on obfuscated malware (Suarez-Tangil et al., 2017; Mirzaei et al., 2019; Kim et al., 2018; Wu et al., 2022; Molina-Coronado et al., 2023; Gao et al., 2024), either for the training or evaluation of the detectors, but what about obfuscated goodware? How do detectors behave in the presence of such apps? Other biases include the use of different obfuscation tools or strategies for malware than for goodware, which results in models that associate obfuscation, or the use of a particular obfuscation tool, to a specific class in the data (Aghakhani et al., 2020). Similarly, experiments performed with malware and goodware captured from different periods can cause biases in the detectors as features such as API calls are time-varying (Arp et al., 2022). Also, most of these studies focused on a small set of features, arguing against other types of features without providing any proof Mirzaei et al. (2019), Garcia et al. (2018). All these aspects may justify the good published results and cause contradictions concerning other analyses carried out for ML-based detectors (Bacci et al., 2018; Molina-Coronado et al., 2023). Finally, we also found that most of them do not provide enough details, data or codes to reproduce their systems.

3. Background

This section briefly introduces some basic concepts that are needed to understand the rest of this paper. This includes the structure and content of an Android Application Package (APK) from which static analysis features are extracted, as well as the types of obfuscation techniques than can be applied and their effect in the apps.

3.1. Android apps

Android apps are usually developed in Java or Kotlin.[4] When an app has to meet very strict performance constraints, or interact directly with hardware components, Android allows developers to introduce native components written in C and C++ (i.e., *native code*). An Android app is distributed and installed via an APK, a compressed (ZIP) file containing all the resources needed (e.g., code, images) to firstly execute the app. Fig. 1 shows the internal structure of an APK file.

Every APK must be signed with the private key of the developer. To validate this signature, the APK contains the public certificate of the developer inside the `META-INF` folder. This mechanism guarantees the

[3] https://github.com/rednaga/APKiD

[4] From now on, we will refer to Java code, although the techniques we describe are also valid for apps written in Kotlin.

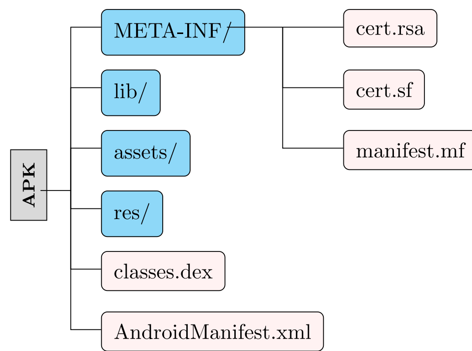


Fig. 1. Structure of an APK file.

integrity of the APK.[5] In a nutshell, before installing an app, Android verifies if the files in the APK match a pre-computed signature and continues with the installation only if the integrity check succeeds.

The `AndroidManifest.xml` defines the structure of an Android app and its meta-data, such as the package name of the app, the required permissions, and the main components (i.e., Activity, Service, Broadcast Receiver, and Content Provider). An Android app can contain one or multiple DEX file(s) (i.e., `classes*.dex`), which include the compiled Java code. Each `.dex` file can reference up to 64k methods (Google Developers, 2020), such as the Android framework methods, other library methods, and the app-specific methods. For the native components, Android provides an Android Native Development Kit (NDK) (Ratabouil, 2015) that generates native libraries in the form of Linux shared objects. Such objects are stored into the `lib` folder.

Finally, the `res` folder contains the compiled resources (e.g., images, and strings), and the `assets` directory includes the raw resources, providing a way to add arbitrary files such as text, HTML, font, and video content into the app.

3.2. Obfuscation

Obfuscation is the process of modifying an executable without altering its functionality (You and Yim, 2010). It aims to counteract automatic or manual code analysis. In the Android context, many strategies can be applied to modify the code or resources within the APK file: from simple operations that change some metadata to bypass basic checks (e.g., signature-based anti-malware), to techniques that explicitly modify the DEX code or resources of the app (Zhang et al., 2021). It is worth emphasizing that in Android obfuscation is more common than in other binary code (e.g., x86 executables), because analyzing and repackaging an Android app is straightforward (Ruggia et al., 2021). In the rest of this Section, we present the type of modifications considered in this work.

Renaming. A DEX file stores the original string-valued identifiers (names) of fields, methods and classes (Google, 2020). Often, these identifiers leak information about code functionalities, lifecycle components and how they interact with each other. For instance, a common practice by programmers is to add "Activity" to each Java class that implements an activity component. The renaming technique replaces these identifiers with meaningless strings, aiming to remove information about the functionality of the app. Consequently, renaming involves modifying the .dex files and the Manifest file (`AndroidManifest.xml`). Note that this technique cannot be applied to methods of the Android lifecycle (e.g., onCreate, onPause) or Android framework components because that would break the execution logic.

Code manipulation. These techniques manipulate the code present in the DEX files to remove useless operations, hide specific API invocations, and modify the execution flow. The main techniques in this category are:

- *Junk code insertion and Reordering (JCIR)* Junk Coder Insertion technique introduces sequences of useless instructions, such as `nop` (i.e., *no-operation* instructions that do nothing). Other strategies transform the control-flow graph (or CFG) of apps by inserting `goto` instructions or arithmetic branches. For example, a `goto` may be introduced in the code pointing to an useless code sequence ending on another `goto` instruction, which points to the instruction after the first `goto`. The arithmetic branch technique inserts a set of arithmetic computations followed by branch instruction that depends on the result of these computations, crafted in such a way that the branch is never taken (Aonzo et al., 2020). This technique also implies Reordering basic blocks in code by inverting branch conditions.
- *Call indirection (CI)* This technique aims to modify the call graph and, therefore, the CFG of the app. It introduces a new intermediate chain of method invocations in the code, adding one or several nodes between a pair of nodes in the original graph. For example, given a method invocation from m_{or1} to m_{or2} in the code, m_{or1} is modified to call to the start of a sequence of n intermediate methods (m_i : $1 <= i <= n$) that end in a call to m_{or2}. In this way, the analysis could not reveal that m_{or2} is actually invoked by m_{or1} (Rastogi et al., 2013).
- *Reflection* This technique uses the reflection capability of the Java language to replace direct method invocations with Java reflection methods that use class and method identifiers as parameters to perform the call. This makes actual method invocations difficult to inspect (Rastogi et al., 2013). Listings 1 and 2 show an example of this transformation. In Listing 1, the method `m1` (of the class `MyObject`) is accessed through the operator "." from the object instance, whereas in Listing 2 shows the same invoked method using the Java reflection API. In this example, a `java.lang.reflect.Method.invoke()` object is created (lines 2–3) and invoked (line 4) for a specific object instance (i.e., `obj`), whereas the class and method names are passed as parameters of these functions.
- *Encryption* This technique prevents accessing to parts or the entire code or resources (e.g., strings and asset files) of the app by using symmetric encryption algorithms. It involves storing the original code or resources in an encrypted form so that a decryption routine, inserted in the code, is invoked whenever an encrypted part needs to be accessed. The decryption key is stored somewhere in the APK or calculated at runtime. This technique introduces extra latency during app execution and severely complicates the analysis of the functionality of the encrypted part (Zhang et al., 2021).

It is worth emphasizing that different obfuscation techniques can be combined to improve their effectiveness. For example, encrypting the strings of reflective calls can hide the method and class names invoked at runtime. This makes it difficult to recover these values by static analysis of the apps. Listing 3 shows an example of the application of both obfuscation techniques to the code in Listing 1. In particular, the class and method names are decrypted at runtime (lines 2–3), hiding which methods are actually invoked. Note how these values are exposed only in an encrypted form, and could change if a different encryption key or algorithm was employed.

4. Dataset

For our experiments, firstly, we constructed a dataset with obfuscated and non-obfuscated apps. From this collection of apps, and by means of static analysis, we obtain a set feature vectors that constitute the object of this study. This section describes how the app dataset is built and the types of features derived from the apps.

[5] Note that Android does not verifies the validity of the developer's certificate but instead, uses this mechanism to validate the integrity of the content within the APK. Therefore, the developers' certificates can be self-signed.

Listing 1 Java direct method invocation

```
1   public class com.example.MyObject implements MyInterface {
2       public Object m1(Object prm) {}
3   }
4
5   public Object standardInvoke(com.example.MyInterface obj, Object arg) {
6       return obj.m1(param);
7   }
```

Listing 2 Java reflective method invocation

```
1   public Object refectionInvoke(com.example.MyInterface obj, Object arg) {
2       Class<?> cls = Class.forName("com.example.MyObject");
3       Method m = cls.getDeclaredMethod("m1");
4       return m.invoke(obj, param);
5   }
```

Listing 3 Java reflective method invocation with encrypted values

```
1   public Object refEncrInvoke(com.example.MyInterface obj, Object arg) {
2   String className = decrypt("AXubduuiao...ZXW");
3   String methodName = decrypt("uibdadBUID...ncu");
4   Class<?> cls = Class.forName(className);
5   Method m = cls.getDeclaredMethod(methodName);
6   return m.invoke(obj, param);
7   }
```

4.1. App dataset

We build our app dataset using a subset of APKs downloaded from the AndroZoo repository (Allix et al., 2016), which contains more than 20 million of APKs with associated meta-data. This meta-data includes the source of the APK, the date, and the number of positive detections (VTD) in VirusTotal. Our objective was to obtain a dataset with the same number of malware and goodware samples, all of them free of obfuscation. We downloaded thousands of samples and filtered out those marked by APKiD[6] as "suspicious" of including obfuscation. To label samples we relied on the VTD values (Zhu et al., 2020): an app with VTD≥7 is considered malware, while an app with VTD=0 was considered goodware (apps with intermediate VTD values were filtered out).

In the second step, we generated obfuscated versions of the apps in the downloaded dataset. To perform this process, we utilized three open-source obfuscation tools: DroidChameleon (Rastogi et al., 2013), AAMO (Preda and Maggi, 2017), and ObfuscAPK (Aonzo et al., 2020). These tools were selected for their wide range of available obfuscation techniques, their open-source nature, and their demonstrated ability to evade Android malware detectors in prior studies. Specifically, for each obfuscation tool, we try to obfuscate every app in the dataset using six obfuscation strategies: Renaming, Junk Code Insertion and Reordering, Reflection, Call Indirection and Encryption. Table 1 summarizes the specific techniques under each strategy. The tools were configured using their default settings for all techniques. The results of this process are summarized in Table 2.

Note that some tool combinations failed due to errors during the APK decompilation/compilation process. It is worth noticing that there were more failures in the case of malware apps than in goodware apps. ObfuscAPK was the tool with the best success rate, correctly obfuscating an average of 85% of the apps. On the contrary, we were unable to obtain obfuscated samples when trying to apply Encryption with AAMO, due to bugs introduced in the code by this tool that prevent the APK from being rebuilt. The attempts to use Renaming with DroidChameleon were also unsuccessful due to an error in the implementation of the tool. For other techniques, DroidChameleon and AAMO had average success rates of 55% and 28%, respectively. We hypothesize that apps which could not be successfully obfuscated by any of the tools are either already obfuscated or face issues with the obfuscation tools themselves, potentially due to bugs in the tools or incompatibilities with certain app characteristics. We therefore decided to exclude these apps from the dataset to avoid introducing potential false negatives into the analysis. After completing the obfuscation process, we compiled a final "Clean" dataset which consists of 4749 goodware and 4067 malware samples that succeeded to be obfuscated

Table 1
List of obfuscation techniques applied to samples.

Category	Obfuscation techniques
Renaming	Class Rename, Field Rename, Method Rename
Junk Code Insertion and Reordering	Arithmetic Branch Insertion, Goto Insertion (unconditional jumps), Reorder (branch inversion)
CallIndirection	Function Call Indirection
Reflection	Reflection, Advanced Reflection
Encryption	Asset Encryption, String Encryption, Native Code Encryption

Table 2
Success rate of different technique-tool obfuscation combinations for the apps in the Clean dataset. The first part of the name refers to the tool used to obfuscate the apps, with *DC* for DroidChamaleon, *AA* for AAMO, and *OA* for ObfuscAPK. The characters after the underscore refer to the strategy followed to obfuscate the apps: renaming (*rnm*), junk code insertion and reordering (*jcir*), call indirection (*ci*), reflection (*refl*) and encryption (*encr*).

Tool-technique	#Goodware samples	#Malware samples	Obf. success rate
DC_rnm	–	–	0%
AA_rnm	2244	1953	34%
OA_rnm	5690	4317	81%
DC_jcir	1855	1123	24%
AA_jcir	2289	2019	35%
OA_jcir	6003	4755	87%
DC_ci	3664	2209	47%
AA_ci	1337	1362	22%
OA_ci	6050	4765	87%
DC_refl	6200	3993	82%
AA_refl	1332	1402	22%
OA_refl	6080	4802	88%
DC_encr	5008	3746	70%
AA_encr	–	–	0%
OA_encr	6074	4814	88%

[6] https://github.com/rednaga/APKiD

Table 3
Composition of datasets used in this work. The columns indicate the number of samples that comprise each set. The CleanSuccObf dataset contains the clean (original) apps for which we obtained obfuscated versions with all tools for at least one technique.

Dataset	#Goodware samples	#Malware samples
Clean	4749	4067
NonObf	1345	1211
CleanSuccObf	3404	2856
Renaming	3238	2868
JCIR	1515	1008
CallIndirection	2118	1737
Reflection	2667	2484
Encryption	4790	4060

with at least one technique and tool. Additionally, we created 14 different datasets, each corresponding to a specific tool-technique combination, with varying compositions of obfuscated apps derived from the "Clean" dataset.

Table 3 summarizes the different datasets that will be used in the experiments. The criteria for the composition of these datasets will be explained in Section 5.

- NonObf: It includes the non obfuscated versions of the apps in the Clean dataset for which we could not obtain an obfuscated version with all the tools for at least one technique, i.e., apps that can be obfuscated using a specific tool and technique but not with the remaining tools using the same technique.
- CleanSuccObf: includes the subset of non obfuscated apps present in Clean, but not in NonObf. That is, the apps for which all the tools have worked for at least one technique.
- The remainder datasets (Renaming, JCIR, CallIndirection, Reflection, and Encryption) contain the obfuscated versions of the apps in CleanSuccObf for that particular technique.

4.2. Feature dataset

An app dataset has to be transformed into a dataset of feature vectors prior to perform malware detection using ML. Following a detailed literature analysis, we identified seven families of static analysis features that have proven to be useful for ML-based malware detection (Wang et al., 2019). We used two well-known and widely used static analysis frameworks for Android to extract these features: Androguard (Desnos et al., 2018) and FlowDroid (Arzt et al., 2014). Sources of these features include: the *classes.dex* and *AndroidManifest.xml* files, as well as the contents of the *res* and *assets* directories of APKs.

4.2.1. Permissions

Permissions have commonly been used as a source of information for malware detection in Android (Arp et al., 2014; Wang et al., 2014; Feizollah et al., 2017; Zhu et al., 2018). In this category, we consider as features the full set of permissions provided by Google in the Android documentation,[7] as well as the set of custom[8] permissions that developers may declare to enforce some functionality in their apps. Following this procedure, we extracted a set of 683 binary features, each corresponding to the presence or absence of a given permission.

4.2.2. Components

An app consists of different software components that must be declared in the *AndroidManifest.xml* file. These elements have been widely used as a source of information for malware detectors (Wu et al., 2012; Arp et al., 2014; Xu et al., 2016; Feizollah et al., 2017). We extract a list of hardware and software components that can be declared using the <*uses-feature*>tag from the Android documentation,[9] as well as every identifier for Activity, Service, ContentProvider, BroadcastReceivers and Intent Filters. In total, we obtained a set of 85 476 binary features, whose value is set to True or False for an app according to the presence of the feature in its *AndroidManifest.xml* file. We additionally derive seven frequency features accounting for the number of elements of each type in the app.

4.2.3. API functions

API libraries allow developers to easily incorporate additional functionality and features into their apps, being the main mean of communication between the programming layer and the underlying hardware. As such, analyzing the calls to methods of these libraries (API functions) constitutes a good instrument to characterize the functionality of apps, and, therefore, for malware detection. Following similar approaches to those proposed in the literature (Wu et al., 2012; Arp et al., 2014; Zhu et al., 2018; Koli, 2018), we extract a binary feature for each API method, and set its value to True if the app contains any call to that method within its code. In total, this set consist of 66 118 binary features.

4.2.4. Opcodes

The compiled Android code (Dalvik) consists of a sequence of opcodes. Opcode-based features provide insights about the code habits of developers as they represent fine-grained information about the functionality of apps (Kim et al., 2018). Subsequences of opcodes, or simply n-grams, have been used for Android malware detection in Jerome et al. (2014), Canfora et al. (2015), Kang et al. (2016), McLaughlin et al. (2017). Concerning the size of the subsequences, Jerome et al. (2014) and Canfora et al. (2015) observed that $n = 2$ offers a good trade-off between the size of the feature vector generated and the performance obtained by detectors. Therefore, we extract unique opcode subsequences of length 2 (or bi-grams) from the code of the apps, and create a feature to represent the number of appearances of each bigram in the code. The resulting vector contains a total of 25 354 frequency features.

4.2.5. Strings

The APK file strings are a valuable source of information for malware detection. In this regard, the most common strings include IP addresses, host names and URLs (Arp et al., 2014; Wang et al., 2017); command names (Yerima et al., 2013; Zhang and Jin, 2016) and numbers (Wang et al., 2017). We processed app files and found 2 425 892 unique strings. Following the procedure in Aghakhani et al. (2020), we observed that 98.5% of the strings were present in less than 1% of the samples. After removing these rare strings, we obtained 39 793 binary features, each representing the presence or absence of a specific string within the app files.

4.2.6. File related features

This type of features includes the size of code files and different file types inside the APK (Grace et al., 2012; Yerima et al., 2013; Wang et al., 2017; Suarez-Tangil et al., 2017). We base our file type extractor on both, the extension of the file and the identification of the first bytes of the content (i.e., magic numbers) of files. The result is a new frequency feature for every unique combination of the extension (*ext*) and magic type (*mtype*), identified as *ext_mtype*. For files without extension, we use the complete file name instead. In total, this set consist of 65 986 frequency features per app.

[7] https://developer.android.com/reference/android/Manifest.permission
[8] https://developer.android.com/guide/topics/permissions/defining
[9] https://developer.android.com/guide/topics/manifest/uses-feature-element.html

Table 4
Persistence of static analysis features when comparing clean and obfuscated apps using ObfuscAPK (OA), DroidChameleon (DC) and AAMO (AA).

	Renaming			Junk Code Insertion and Reordering			Call Indirection			Reflection			Encryption			Avg.
	OA	DC	AA	OA	DC	AA	OA	DC	AA	OA	DC	AA	OA	DC	AA	
Permissions	0.972	–	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.932	0.799	1.0	1.0	1.0	–	0.977
Components	0.219	–	0.999	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	–	0.939
API functions	0.480	–	0.717	1.0	1.0	1.0	0.493	0.718	0.489	0.985	0.407	0.487	0.994	0.999	–	0.751
Opcodes	0.985	–	0.993	0.269	0.107	0.116	0.832	0.832	0.970	0.979	0.919	0.826	0.959	0.982	–	0.751
Strings	0.995	–	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.906	1.0	0.897	0.028	0.009	–	0.833
File Related	0.895	–	0.993	0.803	0.880	0.987	0.791	0.874	0.983	0.845	0.904	0.987	0.926	0.923	–	0.907
Ad-hoc	0.923	–	0.942	0.655	0.345	0.560	0.455	0.667	0.409	0.888	0.859	0.850	0.92	0.922	–	0.722

4.2.7. Ad-hoc features

As explained earlier, some specific detectors claim to use obfuscation-resistant features. We call the features used by these detectors that do not fall into any of the above categories ad-hoc features. They include: semantic features based on sink and source relationships in the code (Zhang and Jin, 2016); certificate information (Suarez-Tangil et al., 2017); flags about the use of cryptographic, reflective, and command execution classes (Grace et al., 2012; Wang et al., 2017; Koli, 2018); and resolved function names for native and reflective calls (Garcia et al., 2018). Due to the computational cost of obtaining these features, we limited the time spent computing them to 15 min per sample. The result is a set of 35 387 frequency features, each representing the number of occurrences of the feature within the app.

5. Feature validity

As a first step in this study, we have designed a set of experiments to determine the robustness and detection ability of the seven feature families described in the previous section when obfuscation is present. The first experiment analyzes the impact that different obfuscation strategies and tools have on the information described by the features. In the second experiment we evaluate the performance and stability of ML algorithms when using these features for malware detection.

5.1. Feature persistence

In this experiment, we aim to examine the impact of obfuscation on the features presented above. We analyze when and how much the features change in the presence of obfuscation. We highlight the disparities among obfuscation tools and how different implementations strategies of the same obfuscation objective can affect the features.

To analyze these aspects, we calculate the feature *persistence* for each tool-technique obfuscation combination. This is done by determining the average level of overlap between the features of an original (clean) app and its obfuscated counterparts. To compute the feature overlap, we compare each pair of feature vectors calculated for an original app and its obfuscated version, and quantify the proportion of features with exact value matches. Note that for binary-featured representations (Permissions, Components, Strings and API functions), this is equivalent to computing the Jaccard index that measures the ratio between the shared elements and the total number of elements in the union of both feature vectors. Note also that, for frequency vectors, an increment or decrease in one unit or ten units has the same effect in this metric.

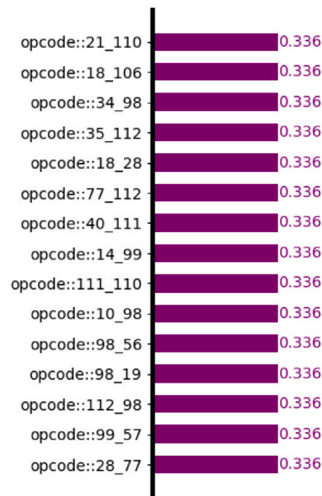
The results of this experiment are shown in Table 4. We find various degrees of persistence, in most cases over 0.8, with many exact matches between the feature vectors of clean and obfuscated APKs. Components and Permission features suffer the smallest changes when applying strategies such as Junk Code Insertion and Reordering, Call Indirection, Reflection and Encryption (independently of the tool). Despite they are affected by all techniques, File-Related features are also among the least affected on average. On the contrary, Ad-hoc, API functions and Opcode feature vectors change the most when obfuscation is applied. However, the average persistence values for these features indicate that most fields (about 75%) are not affected by obfuscation. Therefore, in most cases, we conclude that the use of obfuscation is not reflected as a radical change in the feature vectors.

Persistence values refer to the proportion of features that remain unchanged, but do not tell us which particular features change the most when a tool-technique combination is applied. To shed some light on this regard, we selected the 15 features that change the most when obfuscation is applied. They may belong to different families. To obtain them, we measured the degree of discrepancy in the number of occurrences of each of these features, comparing the original application and the obfuscated version. To simplify the visualization, we show the results for each technique, averaging the discrepancy values for the three tools. The obtained rankings are shown in Fig. 2. The name of each bar is the feature name (which includes its family). The number at the right of each bar is the degree of discrepancy, i.e., the average difference in the frequency of the feature between original and obfuscated versions of apps. Note that for easier interpretation, the scales are specific to each figure.

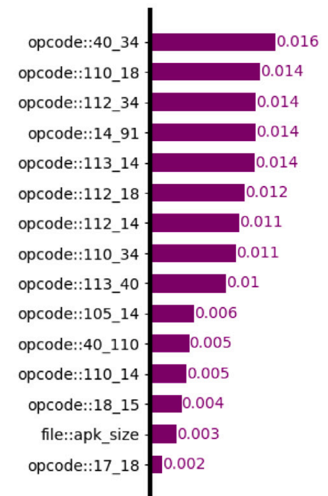
Regarding the persistence of the different feature families, Renaming mainly affected Components and API functions features, due to changes in the names of user-defined packages, classes, methods and fields. It also alters the declaration of custom permissions present in the code, since they depend on the name of the class where they are declared. However, as can be seen in Fig. 2(a), none of these features are among the 15 most affected, mainly because the names assigned to the classes are app-specific. In contrast, Opcode features are among those most significantly affected, due to changes in the order of methods when processing class files. This mainly changes the frequency of sequences that present invocation instructions (opcodes 110, 111 and 112).

In concordance with persistence values in Table 4, Figs. 2(b) and 2(c) show that Junk Code Insertion and Reordering, and Call Indirection techniques are particularly detrimental for features based on code information, with Opcode and Ad-hoc features being the most sensitive to both types of obfuscation. In particular, Ad-hoc features are the most affected by Call Indirection (see Fig. 2(c)) due to the added complexity in the analyses required for their extraction. This is the case of sink and source relations between API functions such as *Cursor.getString* and *Log.i*. Also, due to the addition of indirect calls, this technique increases the frequency of some opcode sequences such as "90_110" formed by an *iput* (90) instruction followed by an invoke*invoke* (110). This technique involves adding hundreds of auxiliary (indirect caller) methods per class, either in separate or in the API classes inside the API. However, these methods are randomly named, which limits their impact (their popularity will be low). As shown in Fig. 2(b), Junk Code Insertion and Reordering greatly alters the frequency of most Opcode sequences due to the inclusion of useless instructions, mainly *goto* (40) and *invoke* (110, 112, 113). The introduction of useless code also greatly impacts on the size of the APK file (File-related feature *file:apk_size*).

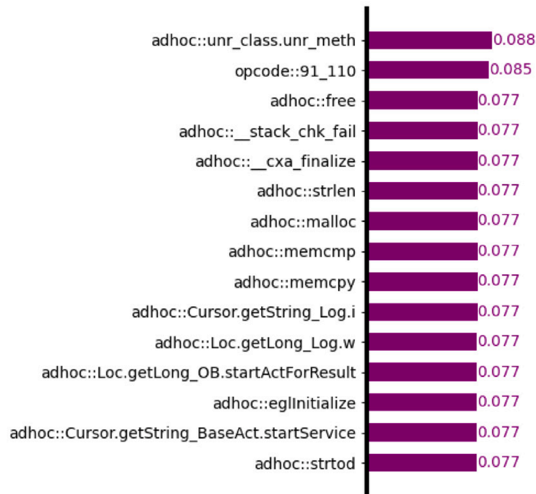
Reflection changes the persistence of features extracted from code analysis. This effect is clearly perceptible in Fig. 2(d). With this technique, the code is modified to hide the originally called methods



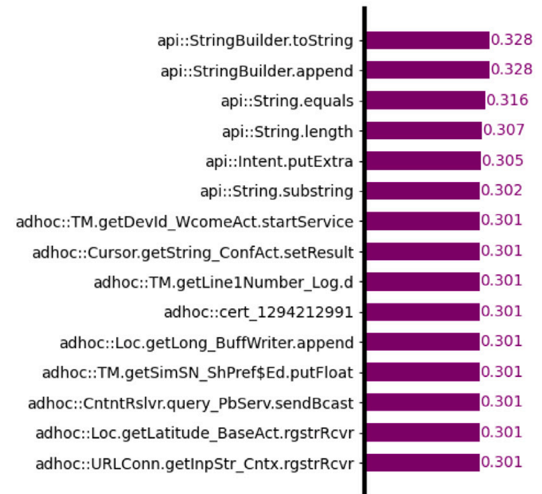
(a) Renaming



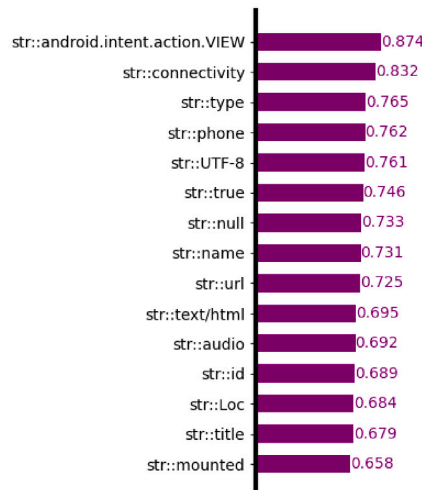
(b) Junk Code Insertion and Reordering



(c) Call Indirection



(d) Reflection



(e) Encryption

Fig. 2. Top 15 of most changed features for each obfuscation strategy. The values on the right indicate the disparity or difference in the frequency of features between the obfuscated apps and their original versions. Results from the three tools have been averaged.

and use reflective calls instead. This mainly affects the API functions that are called more frequently in the code, including string-related functions such as *toString*, *append*, *equals*, or *length*. Ad-hoc functions are among the most changed due to the added complexity of identifying sink and source relationships that contain reflective code. Reflection also results in new string features that contain the class and function names invoked by reflection. However, these are declared once in the code, so their frequency is kept low. Permission features are affected because Reflection can hide the presence of protected API functions that require specific permissions to be granted.

Encryption adds helper classes with the decryption routines that are used to hide user-defined strings and parameters. Therefore, API, Opcode, Ad-hoc, and File-related features are affected by the modifications introduced in the code. However, the main target of this technique are String features, as illustrated in Fig. 2(e) and Table 4. These are heavily affected because their original values are encrypted. In this regard, the top 15 features most changed by encryption are strings related to the app's user interface (`UTF-8`,`phone`, `id`, `title`, `type`).

5.1.1. Differences between obfuscation tools

As seen in Table 4, changes in features depend on the tool used. These differences are due to implementation particularities. Indeed, because of these peculiarities, obfuscation can even alter features that are not primary target of the chosen obfuscation technique. Fig. 3 depicts the average level of overlap between the features obtained for the same apps when obfuscated using different tools. Darker colors indicate less overlap in the obtained feature vectors, while lighter colors represent higher agreement. To better explain the differences obtained, we manually examined the code of these obfuscation tools as well as the features obtained from different samples. Due to space limitations and in order to make this paper more readable, we omit very specific implementation details and limit our discussion to the more prominent differences.

We observed that of all the tools analyzed, none of them considered parameter randomization when implementing the different obfuscation techniques except for Junk Code Insertion and Reordering. As such, for a given tool, the extracted feature vectors are only dependent on the input (app data). In other words, given the same input, a particular tool-technique combination will always return the same output. It is worth mentioning that all tools obfuscate (modify) the Android or Java libraries when this type of content is included in the APK, mainly due to poor checks during obfuscation. Since this code is not user-related, such changes may break the execution of the apps.

The largest differences between obfuscation tools are present for API function and Ad-hoc features. This aspect is clearly perceptible in Figs. 3. In the case of Reflection, we noticed that ObfuscAPK and AAMO perform a fine-grained checking when selecting the set of candidate function calls to be transformed, so that errors introduced by obfuscation are minimal. In contrast, DroidChameleon obfuscates calls whose package matches any of the prefixes included in a pre-defined list, without making any additional checks. In consequence, as shown in Figs. 3(a) and 3(c) feature overlap is low since DroidChameleon results in a higher number of transformed API calls with respect to ObfuscAPK and AAMO.

The way in which the files to be transformed are selected is also the explanation behind the differences observed between AAMO and ObfuscAPK with Renaming (see Fig. 3(b)). By default, ObfuscAPK selects all the files within the APK as candidates for Renaming. This translates into changes in the content of files even if they belong to the Java library or the Android framework. Hence, features in the Components and API functions families are greatly modified by this tool. In contrast, AAMO performs some additional checks aiming to avoid modifying this type of content and presents a reduced impact on these features. Nonetheless, as shown by the persistence values for API-related features in Table 4, these checks are insufficient. For example, classes that are part of the "com.android" package are obfuscated because they do not match the name of the AAMO blacklisted "android" package.

Table 5
Performance of static analysis features for malware detection using non-obfuscated apps for both training and evaluation. TPR stands for the True Positive Rate, i.e., the number of malware correctly identified. FPR stands for the False Positive Rate, i.e., the number of goodware erroneously identified as malware. The A_{mean} is the average of the TPR and the True Negative Ratio (1-FPR).

	TPR	FPR	A_{mean}
Permissions	0.867	0.156	0.855
Components	0.808	0.157	0.825
API functions	0.928	0.081	0.923
Opcodes	0.884	0.252	0.816
Strings	0.907	0.082	0.912
File Related	0.265	0.197	0.534
Ad-hoc	0.768	0.143	0.812

The disparities observed for API function features with CallIndirection between the three tools (see Figs. 3(a)–3(c)) are due to the way intermediate methods are created. ObfuscAPK and AAMO insert the code of intermediate methods within the class file of the original calling method, whereas DroidChameleon adds this code to a (separate) helper class. As a result, methods inserted by ObfuscAPK and AAMO inside the class files of the Android framework are considered as API features by the feature extraction process. Moreover, since these tools use different naming conventions for these new methods, the resulting features do not overlap.

When applying encryption technique, ObfuscAPK and DroidChameleon use different algorithms and parameters. This explains the differences observed between both tools in Fig. 3(a). In particular, ObfuscAPK uses the AES algorithm for encryption, whereas DroidChameleon uses Caesar's algorithm. In both cases, the encryption key is hardcoded in their respective code.

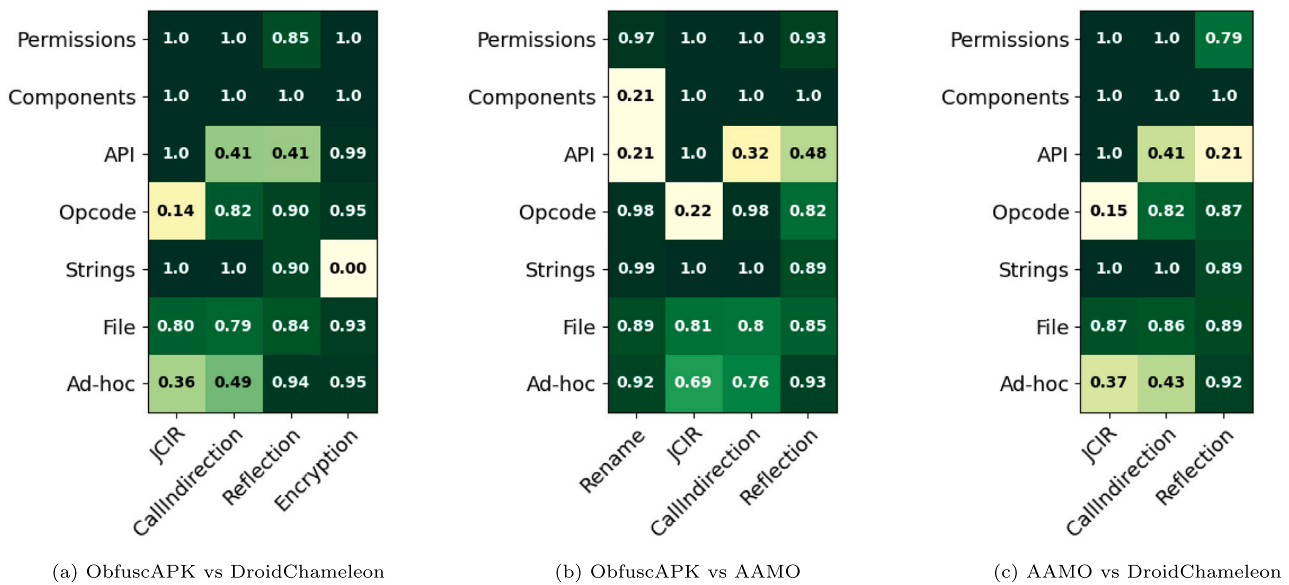
5.2. ML performance

We devise a second set of experiments to (1) analyze the ability of static features to detect malware, and (2) study the stability of these features for malware detection within ML algorithms in the presence of obfuscation. In all experiments, the RandomForests classification algorithm is used without any parameter optimization (Breiman, 2001), as implemented in scikit-learn, a widely-used python library for ML (Pedregosa et al., 2011).

The first scenario is focused on analyzing the predictive power of the different feature families in a fully non-obfuscated (clean) environment. For model training we use the NonObf dataset.[10] For evaluation purposes we used the apps from the CleanSuccObf dataset. Our objective is to evaluate the ability of an off-the-shelf classifier to approximate the class y of apps (malware or goodware) as a function of the original (non-obfuscated) features x_{orig} obtained from clean apps.

Table 5 shows the performance of the trained models. As can be seen, most features present high true positive rates (TPR above 0.8) and moderately low false positive rates (FPR below 0.2). Therefore, it can be concluded that most feature families provide enough information to enable effective malware detection using ML algorithms. This is particularly true for API functions and String features. On the contrary, the model generated using File-Related features performs similar to a random choice model (an A_{mean} value of 0.5) and therefore, we can say that these features are not suitable for the purpose at hand.

[10] Note that an error during the obfuscation process of an app from this set for a given tool can be due to an error in the obfuscation tool, since the same app has been successfully obfuscated using other tools for the same and other strategies.



(a) ObfuscAPK vs DroidChameleon (b) ObfuscAPK vs AAMO (c) AAMO vs DroidChameleon

Fig. 3. Feature overlap between every pair of obfuscation tools using different obfuscation strategies. Note that because Rename and Encryption do not work for DroidChameleon and AAMO, respectively, the corresponding columns are omitted.

Table 6
Feature insensitivity, i.e., the overlap between the classifications made by the ML models for original and their obfuscated variants using ObfuscAPK (OA), DroidChameleon (DC) and AAMO (AA).

	Renaming			Junk Code Insertion and Reordering			Call Indirection			Reflection			Encryption			Avg. Over.
	OA	DC	AA	OA	DC	AA	OA	DC	AA	OA	DC	AA	OA	DC	AA	
Permissions	0.986	–	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.934	0.674	1.0	1.0	1.0	–	0.968
Components	0.506	–	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	–	0.961
API functions	0.986	–	0.992	1.0	1.0	1.0	1.0	1.0	1.0	0.946	0.305	0.998	1.0	1.0	–	0.939
Opcodes	0.950	–	0.964	0.446	0.235	0.087	0.899	0.963	0.900	0.922	0.954	0.893	0.923	0.940	–	0.827
Strings	1.0	–	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.296	0.078	–	0.874
File Related	0.027	–	0.805	0.013	0.052	0.384	0.013	0.072	0.380	0.025	0.106	0.575	0.091	0.030	–	0.197
Ad-hoc	0.624	–	0.813	0.706	0.012	0.583	0.434	0.751	0.415	0.621	0.370	0.238	0.720	0.741	–	0.540

Even with high persistence values, small changes in feature vectors can lead to large changes in the performance of an ML algorithm. This may be the case if the small set of changed features is the most informative for a classifier and strongly influences its prediction. Consequently, this second scenario investigates the sensitivity of ML algorithms to the changes induced by feature vector obfuscation.

We use the ML models trained in the previous experiment (i.e., with clean apps from the NonObf set) and compile two separate evaluation sets for each obfuscation strategy. The first set, known as the obfuscated evaluation set, consists of (obfuscated) samples from the corresponding Renaming, JCIR, CallIndirection, Reflection or Encryption datasets. The other set comprises the clean versions of those apps in the obfuscated dataset. By comparing the predictions made by the ML model for the clean and obfuscated versions of the same app, we can assess whether or not obfuscating an app can change the decision made by the ML model. We leverage the Jaccard index to compute the overlap between the predictions for the clean and obfuscated apps, and we refer to this measure as *insensitivity*. Thus, a high level of insensitivity indicates that the predictions made by a model are preserved even when obfuscation is applied to the apps.

The measured insensitivity values are compiled in Table 6. As can be seen, the decisions of ML models for some feature families are consistent among tools and techniques. This is the case, for example, of Permissions, Components and API functions for Junk Code Insertion and Reordering, Call Indirection and Encryption strategies, with a perfect match in the predictions. Opcode and Ad-hoc features, in contrast, are sensitive to all obfuscation techniques independently of the used tool. In other cases, differences between tools measured by the persistence of features are evidenced in the sensitivity of features for a

Fig. 4. Overall feature insensitivity of features against different obfuscation strategies.

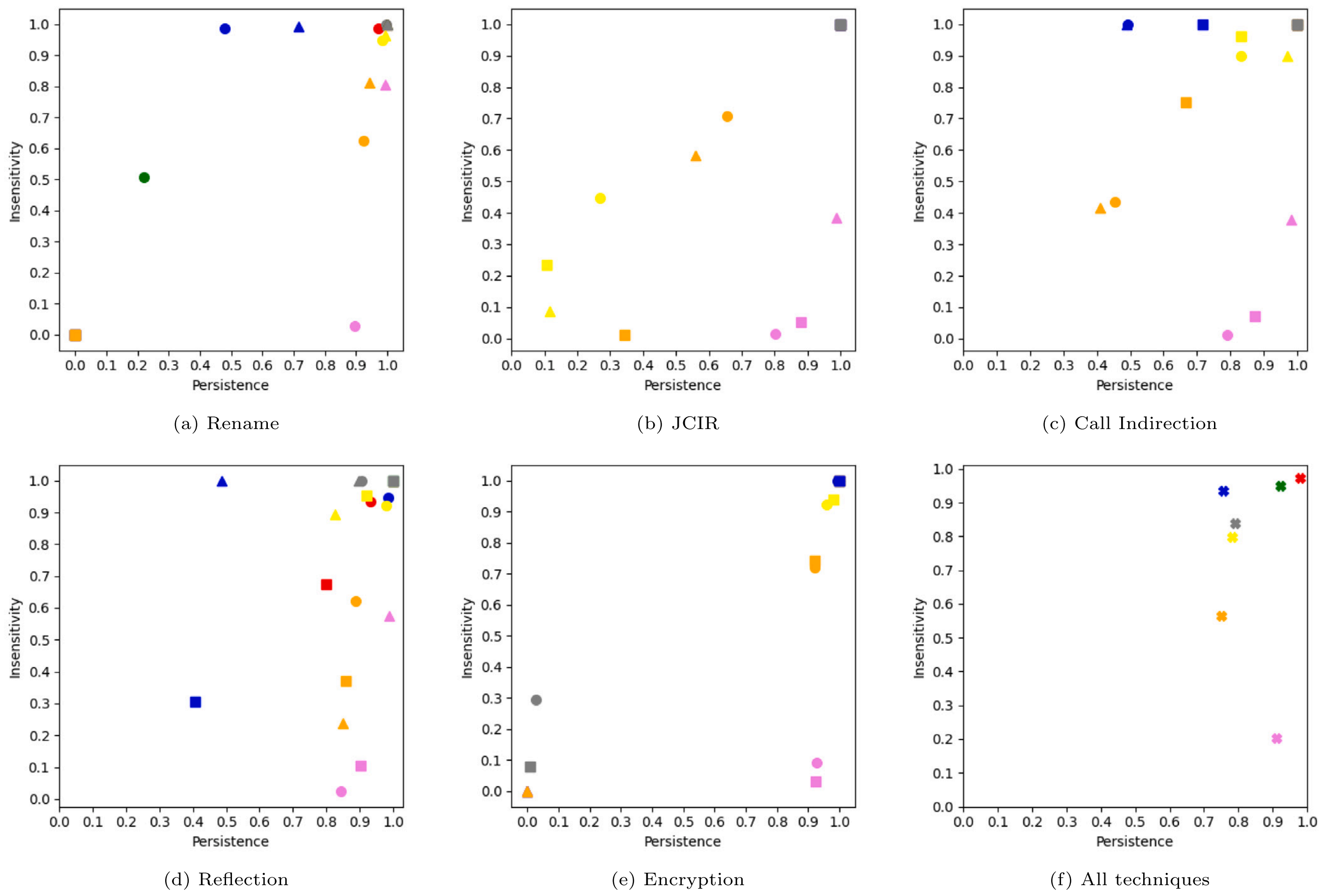


Fig. 5. Relation between persistence and insensitivity to changes of the different features for each obfuscation technique and tool. Every color makes reference to a feature family, with red for Permissions, green for Components, blue for API functions, yellow for Opcodes, gray for Strings, violet for File-Related, and orange for Ad-hoc features; whereas symbols make reference to the values reported on each feature family for ObfuscAPK (circles), AAMO (triangles) and DroidChameleon (squares). The average of all tools is represented by the cross symbol.

specific technique and tool combination. For example, Permissions are more sensitive to reflection when applied using ObfuscAPK or DroidChameleon. Similarly to Permissions and Components when Renaming is applied using ObfuscAPK.

If we observe the average insensitivity values of features independently of the tool used, see Fig. 4, Permissions, Components or API functions result in stable predictions regardless of the obfuscation status of the apps, with insensitivity levels exceeding 0.9 for most strategies. Strings are greatly sensitive to Encryption, with an average value of 0.18, but instead are totally insensitive to any other obfuscation strategy. Similarly, opcode features are highly sensitive to Junk Code Insertion and Reordering but mostly obtained stable predictions for the remaining types of obfuscation. On the contrary, Ad-Hoc, Opcode and File-Related features exhibit high fluctuations in the decisions made by the models for all the strategies. This suggests a greater sensitivity of models to changes introduced by obfuscation in these features.

We wondered if the persistence of features when obfuscation is applied to apps is somehow related to the insensitivity of ML models based on those features. In Fig. 5, we represented the persistence and insensitivity values for the different feature families. As can be seen, in general, there is a high correlation between low persistence and high sensitivity, meaning that larger changes in the features vectors induce larger changes in the predictions of the ML algorithm. See for example Opcode features with JCIR in Fig. 5(b) and String features with Encryption in Fig. 5(e)). However, high persistence values do not necessarily mean that the retained features are the ones that are more helpful to the ML models in making accurate predictions. For example, Ad-hoc features show high persistence values when applying Reflection (changed features are only 16% of the total). Still, the insensitivity is rather low, indicating those include the features that play an important role in the accuracy of predictions (see Fig. 5(d)). Another example is File-related features, which show the most irregular behavior for ML models despite the small proportion of features altered by obfuscation (10% on average as shown in Fig. 5(f)). In this regard, in the previous scenario we evidenced that File-related features lack informativeness for detection (see Table 5).

The previous results highlight an important finding: while persistent features are commonly considered reliable predictors for malware detection, persistence is not the sole factor influencing the robustness of the detection model. On the contrary, high insensitivity values implicate a high persistence on features, so it is a more adequate indicator of robustness. Therefore, it is crucial to carefully examine the impact of obfuscation-induced changes on the informativeness of the features, as even small changes can significantly impact prediction performance. In the next section, we explore the selection of different feature vectors based on ML performance and feature insensitivity to changes to develop robust malware detection models.

6. Robust malware detection

We hypothesize that it is possible to build a robust classifier (one with accurate predictions when dealing with both clean and obfuscated apps) by using features that are both relevant (generate good models with clean apps) and insensitive (the decision of the classifier does not change between the clean and the obfuscated version of an app). We call these *robust* features. On the contrary, features that obtained low

Table 7
Features selected for robust malware detection based on different thresholds for the A_{mean} and feature insensitivity.

Threshold	Feature types	#Features
0.8	Permissions, API functions, Components, Opcodes, Strings	8 683
0.85	Permissions, API functions, Strings	4 683
0.9	API functions	2 000

Table 8
Performance of different robust feature combinations for ML malware detection. A, stands for the model using exclusively API functions. PAS, refers to proposal using Permissions, API functions and Strings, whereas PACOS uses Permissions, API functions Components, Opcodes and Strings.

	Non-Obfuscated			Obfuscated		
	A	PAS	PACOS	A	PAS	PACOS
TPR	0.928	0.920	0.923	0.858	0.889	0.889
FPR	0.081	0.065	0.068	0.060	0.044	0.056
A_{mean}	0.923	0.927	0.927	0.898	0.922	0.916

insensitivity values (i.e., are highly sensitive) or are irrelevant for ML models are prone to cause fluctuations in the predictions of ML models when obfuscation is used.

To select different sets of robust features based on the previous statements, we use the A_{mean} metric reported in Table 5 and the average feature insensitivity reported in Table 6. Specifically, we set three thresholds (0.8, 0.85, and 0.9) for both metrics as the criteria for combining different families of features and train robust ML-based malware detectors. Table 7 shows the feature families that are combined according to each threshold value. As can be seen, the strictest threshold (0.9) considers only the API functions family (A), while the intermediate threshold (0.85) includes API functions, Permissions, and Strings (PAS). The lower threshold (0.8) expands this to include Permissions, API functions, Components, Opcodes, and Strings (PACOS) for robust detection. Once different families of features are identified for combination to form robust features sets, and given the large number of features in some groups, particularly in PAS and PACOS, we select only the 2000 most relevant features of each family. To do so, we used the feature importance rankings for each family of features as provided by the corresponding RandomForest classifiers trained in the experiments described in Section 5.2. Note that this selection does not apply to the Permissions family, as it only contains 683 features.

To experiment with robust detectors using the different feature combinations identified, we train three malware detectors using the RandomForest algorithm with the apps in the NonObf dataset (which does not contain obfuscated samples). This results in three malware detectors using combinations of robust features: one for A features, another one for PAS features and the third one for PACOS features. For evaluation of these detectors with non-obfuscated goodware and malware, we leverage the CleanSuccObf dataset, whereas for the obfuscated scenario, we use the apps from the Renaming, JCIR, CallIndirection, Reflection and Encryption datasets. The prediction performances of the three detectors are summarized in Table 8. As expected, high performance values for the three detectors are obtained for the experiment without obfuscated apps, with true positive rates over 90% and low ratios of false positives (under 8%). In this regard, PAS and PACOS obtained superior performance values to using only API features.

When tested with obfuscated apps, the model relying solely on API function features (A) showed a 3% reduction in A_{mean} performance compared to using the non-obfuscated scenario, primarily due to the impact of reflection on these features, as discussed in Section 5.2. In contrast, incorporating additional features in PAS and PACOS, provided valuable information for detection in this scenario, outperforming the detector that used only API features. Specifically, the PAS detector achieved a 2% reduction in false positives and a 3% increase in correctly detected malware compared to the performance of the model using only API functions. However, adding Component and Opcode features (PACOS) did not improve performance over PAS, resulting in 2% fewer true positives and a 1% reduction in the number of obfuscated goodware being misclassified. Therefore, the most robust model is PAS, which utilizes Permissions, API functions, and Strings.

For comparison, we also evaluate the performance of our best detector, with PAS features, against RevealDroid, a robust state-of-the-art malware detector (Garcia et al., 2018), and Drebin, a high-performing detector (Arp et al., 2014). Both detectors use their own sets of static analysis features and ML algorithms. RevealDroid features include API function and package counts, native calls extracted from binary executables and function names resolved from reflective and dynamic code loading calls. These account for a total of 59 072 features that are used to train a RandomForest model to perform malware detection. The features used by Drebin comprise declared and requested permissions, app components, host names, IPs, commands and suspicious and restricted API functions. This results in 253 881 binary features that are used to train a linear Support Vector Machine (SVM) classifier.

Table 9, shows the performance of our PAS detector against RevealDroid and Drebin. As can be seen, PAS outperformed both state-of-the-art detectors for the non-obfuscated and obfuscated scenarios. With obfuscated apps, our robust proposal obtained a 1% and a 6% higher detection rate, and 4% and 8% lower malware misclassification levels, with respect to Drebin and RevealDroid. In contrast to RevealDroid and Drebin, which mainly rely on Strings and API features, PAS benefits from the combination of a variate feature vector and hence, it is more robust to different obfuscation strategies. These good numbers evidence that using a small set of Permissions, API functions and Strings, that do not require highly complex extraction or preprocessing mechanisms, is enough to perform malware detection in Android using off-the-self ML algorithms. Moreover, it also demonstrates that these features are robust enough to identify obfuscated malware and goodware even without information about the strategy or tool used to obfuscate apps.

7. Discussion and future work

The experiments carried out in this paper evidence that, as commonly assumed (Ye et al., 2017), static analysis features can be affected by specific obfuscation techniques. On one hand, feature persistence showed that all the feature families are affected by at least one obfuscation technique. Among them, the features obtained from the manifest of applications proved to be the most stable. Nonetheless, and contrary to what is commonly argued (Bakour et al., 2018), our experiments also demonstrate that static analysis features can be a reliable source of information for ML malware detection. In this regard, we observed that some obfuscation strategies can result in additional features while leaving the original features unaltered. For example, this is the effect of CallIndirection in API functions, or Reflection in Strings. In most cases, the impact of obfuscation is limited to less than 20% of all the features derived from the samples (for example, at most 20% of the features are affected by Call Indirection and about 15% are altered by Reflection). An interesting line of research in this regard could be to analyze whether static analysis frameworks have flaws that magnify the impact of obfuscation. This aspect would help developers to improve static analysis tools and also facilitate practitioners to select the most reliable tool.

We also observed that the changes caused by obfuscation on the features vary significantly between different tools, mainly due to implementation particularities. The lack of randomization in the analyzed obfuscation tools makes them produce the same output for the same input value, even for different source apps or executions of the same tool. Despite this way of operation is useful to hide the explicit information provided by, for example a class name, it is insufficient to conceal the intrinsic information, i.e., relationships between features, such as correlations. Consequently, apps that contain a similar characteristic,

Table 9
Performance of robust ML detectors based on static analysis features. PAS refers to our robust detection proposal using Permissions, API functions and Strings.

	Non-Obfuscated			Obfuscated		
	RevealDroid	Drebin	PAS	RevealDroid	Drebin	PAS
TPR	0.856	0.914	0.920	0.832	0.876	0.889
FPR	0.117	0.103	0.065	0.12	0.088	0.044
A_{mean}	0.869	0.905	0.927	0.856	0.893	0.922

when obfuscated using the same tool, will maintain similar relations between the obfuscated values than between the original (unobfuscated) features. Additionally, we found that open-source obfuscators need to improve the implementation of some obfuscation techniques due to the high failure rates they present. Therefore, the proposition and implementation of better obfuscation strategies and tools for Android is a promising research area. Such strategies should be accompanied with evaluations to ensure that the execution of the obfuscated apps is not broken.

The performance analysis of ML models trained with static analysis features revealed that some feature types (families) typically proposed for malware detection (Pan et al., 2020), such as file-related features, are not effective to distinguish malware. The experiment, conducted on non-obfuscated applications, revealed API functions and Strings as the most informative features to identify malware, achieving detection rates of over 90% with low false positive rates of 8%. Through careful analyses about the impact of obfuscation-induced changes on the informativeness of features, we have demonstrated that even changes produced to a small proportion of feature values can have a significant impact on performance. This finding demystifies a common assumption in the Android malware detection field, which is to consider highly persistent features as robust. To address this issue, we proposed to use a more precise indicator of the robustness of features which combines both persistence and ML performance of features.

By combining features that exhibited high insensitivity to changes and presented high ML accuracy, we demonstrated that ML-based malware detection using basic static analysis features can be robust against common obfuscation techniques. Remarkably, this remains true even in scenarios where no knowledge about the obfuscation strategies applied to the apps is assumed, i.e., the obfuscated data is not taken into account during the training process. Under such conditions, we proposed a robust detection approach that outperformed RevealDroid, the current state-of-the-art obfuscation-resilient detector, and Drebin, the best proposal for malware detection in Android according to a recent comparative (Molina-Coronado et al., 2023). Specifically, our detector achieved 92% of correct classifications, compared to 89% and 85% for Drebin and RevealDroid, respectively. Accordingly, our focus on obtaining richer and more robust app representations showed that performance on Android malware detection can be improved without considering features that require high computations or relying on extremely complex ML algorithms. Therefore, we believe that further research efforts should shift from ML-centered solutions to the exploration of the different sources of static analysis features for malware detection.

As a final note, we are aware that some limitations apply to the work carried out for this paper. The main one is that our analysis is limited to individual obfuscation strategies. However, obfuscation strategies can be combined in order to increase the probability of circumventing detectors. It should be noted that the number of possible combinations is extremely high. Especially, because the number of techniques that can be combined and the order in which they are applied to apps determines the obfuscation results, i.e., the information hidden by a previous obfuscation technique becomes invisible for the next obfuscation strategy. The cost of assuming such experimentation becomes unfeasible in the context of this work since: (1) samples would have to be obfuscated combining strategies and tools, and (2), feature extraction, comparison and model training would have to be performed for the resulting obfuscated samples. In contrast to such extensive analysis, our study of individual techniques aimed to better understand their impact in the information that is extracted for ML malware detection, as well as to evidence implementation pitfalls and particularities among obfuscation tools. In this regard, this work can be seen as a first step at understanding the impact that the combination of different obfuscation strategies can have on static analysis features. As for future work, we plan to extend our experiments to additional obfuscation techniques, such as packing.

8. Conclusions

This paper explored the effectiveness of static analysis features for ML-based Android malware detection in the presence of obfuscation. To perform this assessment, we generated a variety of datasets by applying different obfuscation strategies to apps with the help of three state-of-the-art open-source obfuscators. Seven families of static analysis features were defined and evaluated throughout an extensive set of experiments. We identified which families are more persistent when obfuscation is applied and determined why persistence is not a good indicator of robustness against obfuscation based on their performance for Android malware detection using obfuscated malware. Based on these findings, we proposed the use of Permissions, API functions and Strings for robust ML-based malware detection. A stock implementation of the RandomForest classification algorithm using these features was used. The generated ML detector is able to separate malware from goodware with a remarkable success rate, without any prior knowledge of the specific obfuscation techniques applied to apps. In particular, this detector correctly identified 89% of evasion attempts with a low false positive rate of 4%, outperforming the current state-of-the-art solution for obfuscation-resistant Android malware detection.

CRediT authorship contribution statement

Borja Molina-Coronado: Writing – review & editing, Writing – original draft, Validation, Software, Project administration, Methodology, Investigation, Data curation, Conceptualization. **Antonio Ruggia:** Writing – original draft, Data curation. **Usue Mori:** Writing – review & editing, Supervision, Funding acquisition, Conceptualization. **Alessio Merlo:** Writing – review & editing, Supervision. **Alexander Mendiburu:** Writing – review & editing, Supervision, Funding acquisition. **Jose Miguel-Alonso:** Writing – review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Borja Molina-Coronado reports financial support was provided by Basque Government. Usue Mori reports financial support was provided by Basque Government. Alexander Mendiburu reports financial support was provided by Basque Government. Jose Miguel-Alonso reports financial support was provided by Basque Government. Borja Molina-Coronado reports financial support was provided by Spain Ministry of Science and Innovation. Usue Mori reports financial support was provided by Spain Ministry of Science and Innovation. Alexander Mendiburu reports financial support was provided by Spain Ministry of Science and Innovation. Jose Miguel-Alonso reports financial support was provided by Spain Ministry of Science and Innovation.

Acknowledgments

This work has received support from the following programs: PID2019-104966GB-I00AEI (Spanish Ministry of Science and Innovation), IT-1504-22 (Basque Government, Spain), KK-2022/00106 (Elkartek project supported by the Basque Government, Spain). Borja Molina-Coronado holds a predoctoral grant (ref. PRE_2021_2_0230) by the Basque Government, Spain.

Data availability

No data was used for the research described in the article.

References

Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., Kruegel, C., 2020. When malware is packin'heat; limits of machine learning classifiers based on static analysis features. In: Network and Distributed Systems Security (NDSS) Symposium 2020.

Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y., 2016. Androzoo: Collecting millions of android apps for the research community. In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories. MSR, IEEE, pp. 468–471.

Aonzo, S., Georgiu, G.C., Verderame, L., Merlo, A., 2020. Obfuscapk: An open-source black-box obfuscation tool for android apps. SoftwareX 11, 100403.

Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., Rieck, K., 2022. Dos and don'ts of machine learning in computer security. In: Proc. of the USENIX Security Symposium.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C., 2014. Drebin: Effective and explainable detection of android malware in your pocket.. In: Ndss. 14, pp. 23–26.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P., 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices 49 (6), 259–269.

Bacci, A., Bartoli, A., Martinelli, F., Medvet, E., Mercaldo, F., Visaggio, C.A., 2018. Impact of code obfuscation on android malware detection based on static and dynamic analysis. In: ICISSP. pp. 379–385.

Bakour, K., Ünver, H.M., Ghanem, R., 2018. The android malware static analysis: techniques, limitations, and open challenges. In: 2018 3rd International Conference on Computer Science and Engineering. UBMK, Ieee, pp. 586–593.

Breiman, L., 2001. Random forests. Mach. Learn. 45, 5–32.

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

Collberg, C.S., Thomborson, C., 2002. Watermarking, tamper-proofing, and obfuscation-tools for software protection. IEEE Trans. Softw. Eng. 28 (8), 735–746.

Desnos, A., Gueguen, G., Bachmann, S., 2018. Androguard. [Online] Available: https://androguard.readthedocs.io/en/latest/.

Dong, S., Li, M., Diao, W., Liu, X., Liu, J., Li, Z., Xu, F., Chen, K., Wang, X., Zhang, K., 2018. Understanding android obfuscation techniques: A large-scale investigation in the wild. In: International Conference on Security and Privacy in Communication Systems. Springer, pp. 172–192.

Feizollah, A., Anuar, N.B., Salleh, R., Suarez-Tangil, G., Furnell, S., 2017. Androdialysis: Analysis of android intent effectiveness in malware detection. Comput. Secur. 65, 121–134.

Gao, C., Cai, M., Yin, S., Huang, G., Li, H., Yuan, W., Luo, X., 2023. Obfuscation-resilient android malware analysis based on complementary features. IEEE Trans. Inf. Forensics Secur..

Gao, C., Huang, G., Li, H., Wu, B., Wu, Y., Yuan, W., 2024. A comprehensive study of learning-based android malware detectors under challenging environments. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. pp. 1–13.

Garcia, J., Hammad, M., Malek, S., 2018. Lightweight, obfuscation-resilient detection and family identification of android malware. ACM Trans. Softw. Eng. Methodol. (TOSEM) 26 (3), 1–29.

Google, 2020. Dalvik executable format. URL https://source.android.com/docs/core/runtime/dex-format. (Accessed online: 17 December 2024).

Google Developers, 2020. Enable multidex for apps with over 64k methods. URL https://developer.android.com/studio/build/multidex. (Accessed online: 17 December 2024).

Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X., 2012. Riskranker: scalable and accurate zero-day android malware detection. In: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services. pp. 281–294.

Hammad, M., Garcia, J., Malek, S., 2018. A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In: Proceedings of the 40th International Conference on Software Engineering. pp. 421–431.

Hastie, T., Tibshirani, R., Friedman, J., 2009. The elements of statistical learning: data mining, inference, and prediction. Springer Science & Business Media.

Jerome, Q., Allix, K., State, R., Engel, T., 2014. Using opcode-sequences to detect malicious android applications. In: 2014 IEEE International Conference on Communications. ICC, IEEE, pp. 914–919.

Kang, B., Yerima, S.Y., McLaughlin, K., Sezer, S., 2016. N-opcode analysis for android malware classification and categorization. In: 2016 International Conference on Cyber Security and Protection of Digital Services (Cyber Security). IEEE, pp. 1–7.

Kaspersky Labs, 2021. Mobile malware evolution 2020. [Online] Available: https://securelist.com/mobile-malware-evolution-2020/101029/.

Kim, T., Kang, B., Rho, M., Sezer, S., Im, E.G., 2018. A multimodal deep learning method for android malware detection using various features. IEEE Trans. Inf. Forensics Secur. 14 (3), 773–788.

Koli, J., 2018. RanDroid: Android malware detection using random machine learning classifiers. In: 2018 Technologies for Smart-City Energy Security and Power. ICSESP, IEEE, pp. 1–6.

Lee, W.Y., Saxe, J., Harang, R., 2019. SeqDroid: Obfuscated android malware detection using stacked convolutional and recurrent neural networks. In: Deep Learning Applications for Cyber Security. Springer, pp. 197–210.

Li, L., Bissyandé, T.F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., Traon, L., 2017. Static analysis of android apps: A systematic literature review. Inf. Softw. Technol. 88, 67–95.

Maiorca, D., Ariu, D., Corona, I., Aresu, M., Giacinto, G., 2015. Stealth attacks: An extended insight into the obfuscation effects on android malware. Comput. Secur. 51, 16–31.

McLaughlin, N., Martinez del Rincon, J., Kang, B., Yerima, S., Miller, P., Sezer, S., Safaei, Y., Trickel, E., Zhao, Z., Doupé, A., et al., 2017. Deep android malware detection. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy. pp. 301–308.

Mirzaei, O., de Fuentes, J.M., Tapiador, J., Gonzalez-Manzano, L., 2019. AndrODet: An adaptive android obfuscation detector. Future Gener. Comput. Syst. 90, 240–261.

Mohammadinodooshan, A., Kargén, U., Shahmehri, N., 2019. Comment on" AndrODet: An adaptive android obfuscation detector". arXiv preprint arXiv:1910.06192.

Molina-Coronado, B., Mori, U., Mendiburu, A., Miguel-Alonso, J., 2020. Survey of network intrusion detection methods from the perspective of the knowledge discovery in databases process. IEEE Trans. Netw. Serv. Manag. 17 (4), 2451–2479.

Molina-Coronado, B., Mori, U., Mendiburu, A., Miguel-Alonso, J., 2023. Towards a fair comparison and realistic evaluation framework of android malware detectors based on static analysis and machine learning. Comput. Secur. 124, 102996.

Pan, Y., Ge, X., Fang, C., Fan, Y., 2020. A systematic literature review of android malware detection using static analysis. IEEE Access 8, 116363–116379.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E., 2011. Scikit-learn: Machine learning in python. J. Mach. Learn. Res. 12, 2825–2830.

Preda, M.D., Maggi, F., 2017. Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. J. Comput. Virol. Hacking Techn. 13 (3), 209–232.

Rastogi, V., Chen, Y., Jiang, X., 2013. Droidchameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. pp. 329–334.

Rastogi, V., Chen, Y., Jiang, X., 2014. Catch me if you can: Evaluating android anti-malware against transformation attacks. IEEE Trans. Inf. Forensics Secur. 9 (1), 99–108.

Ratabouil, S., 2015. Android NDK: beginner's guide. Packt Publishing Ltd.

Ruggia, A., Losiouk, E., Verderame, L., Conti, M., Merlo, A., 2021. Repack me if you can: An anti-repackaging solution based on android virtualization. In: Annual Computer Security Applications Conference. pp. 970–981.

Sadeghi, A., Bagheri, H., Garcia, J., Malek, S., 2016. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. IEEE Trans. Softw. Eng. 43 (6), 492–530.

Sihag, V., Vardhan, M., Singh, P., 2021. A survey of android application and malware hardening. Comp. Sci. Rev. 39, 100365.

Statista, 2021. Mobile operating systems' market share worldwide from january 2012 to january 2021. [Online] Available: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.

Suarez-Tangil, G., Dash, S.K., Ahmadi, M., Kinder, J., Giacinto, G., Cavallaro, L., 2017. Droidsieve: Fast and accurate classification of obfuscated android malware. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy. pp. 309–320.

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

Wang, W., Wang, X., Feng, D., Liu, J., Han, Z., Zhang, X., 2014. Exploring permission-induced risk in android applications for malicious application detection. IEEE Trans. Inf. Forensics Secur. 9 (11), 1869–1882.

Wang, X., Wang, W., He, Y., Liu, J., Han, Z., Zhang, X., 2017. Characterizing android apps' behavior for effective detection of malapps at large scale. Future Gener. Comput. Syst. 75, 30–45.

Wang, W., Zhao, M., Gao, Z., Xu, G., Xian, H., Li, Y., Zhang, X., 2019. Constructing features for detecting android malicious applications: issues, taxonomy and directions. IEEE Access 7, 67602–67631.
Wu, Y., Dou, S., Zou, D., Yang, W., Qiang, W., Jin, H., 2022. Contrastive learning for robust android malware familial classification. IEEE Trans. Dependable Secure Comput..
Wu, J., Kanai, A., 2021. Utilizing obfuscation information in deep learning-based android malware detection. In: 2021 IEEE 45th Annual Computers, Software, and Applications Conference. COMPSAC, IEEE, pp. 1321–1326.
Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M., Wu, K.-P., 2012. Droidmat: Android malware detection through manifest and api calls tracing. In: 2012 Seventh Asia Joint Conference on Information Security. IEEE, pp. 62–69.
Xu, K., Li, Y., Deng, R.H., 2016. Iccdetector: Icc-based malware detection on android. IEEE Trans. Inf. Forensics Secur. 11 (6), 1252–1264.
Ye, Y., Li, T., Adjeroh, D., Iyengar, S.S., 2017. A survey on malware detection using data mining techniques. ACM Comput. Surv. 50 (3), 1–40.
Yerima, S.Y., Sezer, S., McWilliams, G., Muttik, I., 2013. A new android malware detection approach using bayesian classification. In: 2013 IEEE 27th International Conference on Advanced Information Networking and Applications. AINA, IEEE, pp. 121–128.
You, I., Yim, K., 2010. Malware obfuscation techniques: A brief survey. In: 2010 International Conference on Broadband, Wireless Computing, Communication and Applications. IEEE, pp. 297–300.
Zhang, X., Breitinger, F., Luechinger, E., O'Shaughnessy, S., 2021. Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. Forensic Sci. Int. Dig. Investigat. 39, 301285.
Zhang, X., Jin, Z., 2016. A new semantics-based android malware detection. In: 2016 2nd IEEE International Conference on Computer and Communications. ICCC, IEEE, pp. 1412–1416.
Zhu, S., Shi, J., Yang, L., Qin, B., Zhang, Z., Song, L., Wang, G., 2020. Measuring and modeling the label dynamics of online anti-malware engines. In: 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, pp. 2361–2378.
Zhu, H.-J., You, Z.-H., Zhu, Z.-X., Shi, W.-L., Chen, X., Cheng, L., 2018. DroidDet: effective and robust detection of android malware using static analysis along with rotation forest model. Neurocomputing 272, 638–646.

Borja Molina-Coronado received his M.Sc. in Computer Engineering and his Ph.D. in Computer Science from the University of the Basque Country UPV/EHU, in 2017 and 2023, respectively. He is a researcher in the Dept. of Computer Architecture and Technology of the UPV/EHU. His main research areas are malware analysis, machine learning and network security.

Antonio Ruggia is a Ph.D. student in Security, Risk, and Vulnerability at the University of Genoa since November 2020. He is interested in several security topics, including Mobile Security, with a specific interest in Android, malware, and data protection. He graduated in October 2020 from the University of Genoa and participated in the 2019 CyberChallenge.it, an Italian practical competition for students in Cybersecurity. Since 2018, he has worked as a full-stack developer in a multinational corporation.

Usue Mori received her M.Sc. Degree in Mathematics, and a Ph.D. in Computer Science from the University of the Basque Country UPV/EHU, Spain, in 2010 and 2015, respectively. Since 2019, she has been working as a lecturer in the Dept. of Computer Science and Artificial Intelligence of the University of the Basque Country UPV/EHU. Her main research interests include clustering and classification of time series.

Alessio Merlo received the Ph.D. degree in computer science from the University of Genoa in 2010. He is currently a Full Professor in computer engineering with the Centre for Higher Defence Studies (CASD), Rome, Italy. He has published more than 120 scientific papers in international conferences and journals. His research interests include mobile security, where he contributed to discovering several high-risk vulnerabilities both in applications and the android OS and system security.

Alexander Mendiburu is a full professor at the Dept. of Computer Architecture and Technology of the University of the Basque Country UPV/EHU, where he has been working since 1999. He received his B.Sc. Degree in Computer Science and his Ph.D. Degree from the University of the Basque Country, Spain, in 1995 and 2006, respectively. His main research areas are evolutionary computation, time series, probabilistic graphical models, and parallel computing.

Jose Miguel-Alonso is a full professor at the Dept. of Computer Architecture and Technology of the University of the Basque Country UPV/EHU. Formerly, he was a Visiting Assistant Professor at Purdue University. He received his M.Sc. in Computer Science in 1989 and his Ph.D. in Computer Science in 1996, both from the UPV/EHU. He carries out research related to parallel and distributed systems, in areas such as network security, software security, performance modeling and resource management in large-scale computing systems. Prof. Miguel-Alonso is a member of the IEEE Computer Society and of the HiPEAC European Network of Excellence.