applied sciences

MDPI

Article

Evaluation of Survivability of the Automatically Obfuscated Android Malware

Himanshu Patel, Deep Patel, Jaspreet Ahluwalia, Vaishali Kapoor, Karthik Narasimhan, Harmanpreet Singh, Harmanjot Kaur, Gadi Harshitha Reddy, Sai Sushma Peruboina and Sergey Butakov *

Information Systems Security and Assurance Management, Concordia University of Edmonton, Edmonton, AB T5B 4E4, Canada; hcpatel@student.concordia.ab.ca (H.P.); dpatel6@student.concordia.ab.ca (D.P.); jahluwal@student.concordia.ab.ca (J.A.); vkapoor@student.concordia.ab.ca (V.K.); knarasim@student.concordia.ab.ca (K.N.); hsingh35@student.concordia.ab.ca (H.S.); hkaur19@student.concordia.ab.ca (H.K.); greddy@student.concordia.ab.ca (G.H.R.); speruboi@student.concordia.ab.ca (S.S.P.)

* Correspondence: sergey.butakov@concordia.ab.ca

Featured Application: Findings of the paper can be implemented in malware detection applications.

Abstract: Malware is a growing threat to all mobile platforms and hundreds of new malicious applications are being detected every day. At the same time, the development of automated software obfuscation techniques allows for the easy production of new malware variants even by attackers with entry-level programming skills. Such obfuscation techniques can evade the signature-based mechanism implemented in current antimalware technology. This paper presents the results of a study that examined how automated obfuscation techniques affect malicious and benign applications by two widely used malware detection approaches, namely static and dynamic analyses. The research explored 5000 samples of malware and benign programs and evaluated the impact of automated obfuscation on Android applications. The experimental results indicated that (1) up to 73% of the reviewed applications "survived" the automated obfuscation; (2) automated obfuscation reduced the detection ratio to 65–85% depending on the obfuscation method used. These findings call for a more active use of advanced malware detection methods in commonly used antivirus platforms.

Keywords: malware; software obfuscation; static analysis; dynamic analysis; malware detection

Citation: Patel, H.; Patel, D.; Ahluwalia, J.; Kapoor, V.; Narasimhan, K.; Singh, H.; Kaur, H.; Reddy, G.H.; Peruboina, S.S.; Butakov, S. Evaluation of Survivability of the Automatically Obfuscated Android Malware. *Appl. Sci.* **2022**, *12*, 4969. https://doi.org/10.3390/app12104969

Academic Editor: Gianluca Lax

Received: 8 April 2022
Accepted: 8 May 2022
Published: 14 May 2022

1. Introduction

The Android platform is one of the most popular mobile device platforms, with installation on billions of devices worldwide [1]. As the popularity of smartphones has grown, so has the number of malware applications targeting such devices and alternative Android application repositories that distribute such applications. Consumers typically use anti-malware programs to protect their mobile devices, which scan apps for malicious code. However, these products have not always been able to detect malware. Malware creators frequently rely on code obfuscation to prevent detection. Code obfuscation [2] converts code into a format that is more complicated to decipher, interpret and reverse engineer for humans and computers. Such a modification does not alter the semantics of the code. Code obfuscation may be minor or sophisticated, such as bytecode encryption or the addition of unused code [3]. There are several commercial and open-source obfuscators available on the market, such as Proguard, DexGuard, Obfusapk, etc. [4,5]. They provide the ability to employ single or multiple code obfuscation strategies to the application code to prevent the reverse engineering of code and to protect the proprietary code. However, malware developers leverage the same tools to perform code obfuscation with the malicious code and to inject it inside the benign application to bypass anti-malware tools. The accessibility

of reverse engineering tools in conjunction with rich bytecode semantics has led to an exponential increase in malware for all types of mobile platforms.

Consequently, substantial attempts have been made to establish strategies to identify Android malware. Anti-malware products, based on the detection methodology used, can be classified based on the following two broad categories: static and dynamic detection. Static detection analyzes the application code through reverse engineering techniques without the need to run the Android application (APK). The dynamic detection technique analyzes the application's run time behavior to detect potentially malicious system calls.

This project discusses (1) the effects of single and combined obfuscation techniques on the detection capability of anti-malware products through multiple obfuscation tools, (2) the accuracy of anti-malware products to differentiate between malicious and benign apps after transformation, (3) the impact of time on the identification of individual items by an obfuscated app and (4) the "survival" ratio of malware after being subjected to obfuscation.

1.1. Background

1.1.1. Android Platform

Android supports the Java language and enables developers to build an application using the available Java libraries. The Android architecture consists of the following five layers: application, application framework, libraries and Dalvik virtual machines, Android runtime, and Linux kernel [6]. The Linux kernel handles the functionalities related to storage, power, application and device drivers, network, memory, and process management. The application developer uses the Linux kernel to perform various tasks, ranging from process management to security. While developing an Android application, developers make use of these services to perform the intended activities [7]. The layer that interacts with the end user is an Application, for example, Browser, Music player, Photo album, etc. The security and privacy concerns related to the developed application must be overseen by the application developer while other components of the architecture are used to enforce security settings. Structure of the Android application file is presented in Figure 1 [8]. For example, the androidmanifest.xml file defines the permissions that can be used by an application. This file will be explored in detail in the analysis performed in this research.

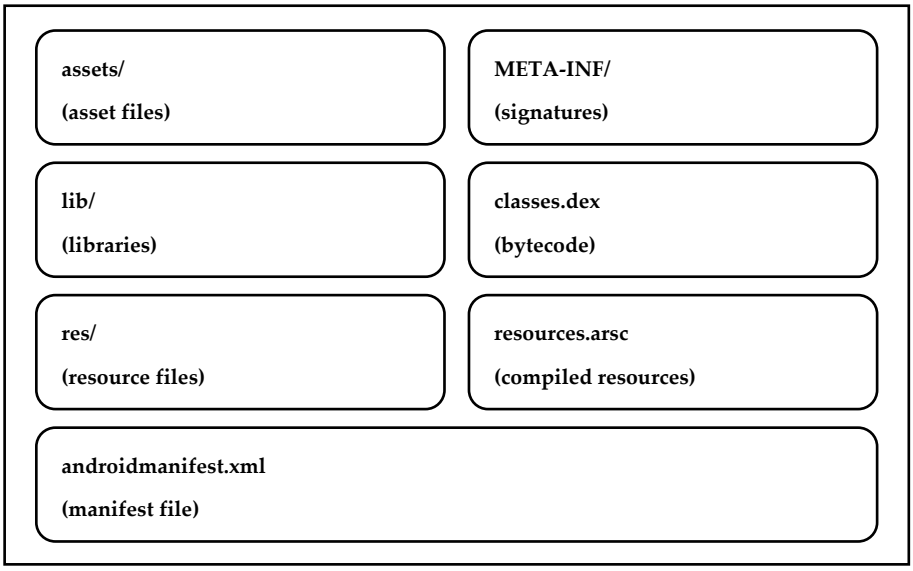

Figure 1. APK file structure.

Applications running on the Android platform can call or use an element of other installed or running applications. This function can be achieved by essential components such as activity, services broadcast receivers, and content providers [9]. The subclass for each activity is written, and each activity is inherited from the activity class, making it

the base class. Services are also considered the main component of any application, and they run in the background while an application is in use. The components of the outlined architecture form the Android attack surface.

An attack surface is a primary attribute used to classify if a specific target is vulnerable to attack based on risk. An attack vector applies to the way an intruder targets a device. In other words, a vulnerable code can be considered an attack surface. Unlike an attack vector, an attack surface does not depend on the attacker's actions or require a vulnerability to exist; instead, it describes the places in the code where vulnerabilities may be present. Additionally, a remote attack surface is one of the most common attack methodologies used by attackers to gain local or root access to the Android terminal [10].

An example of a local attack on the platform can include changes to the permissions specified in the androidmanifest.xml file required by the Android application. APK tempering is a vulnerability that, if exploited, can be mitigated by adding an application code-signing mechanism [11]. The Android OS allows developers to sign their applications using a certificate provided by the company that developed the application. After an application is signed, the certificate is used to identify the application, and during communication between the application and the other applications, trust between the two is established.

1.1.2. Malware Analysis

A malware analysis is the process of analyzing malware and studying the components and behavior of malware. The commonly used malware analysis techniques are static and dynamic analysis [12]. Static analysis is a process in which the analysis is performed without running the malware, and it is also more secure when compared to dynamic analysis. In contrast, dynamic analysis is a process of analyzing the malware by running the code, and typically the process should be performed in a more secure/isolated environment [13].

Static analysis is a technique that involves viewing the APK file without inspecting the impact of actual instructions. This type of analysis can verify whether the data is malicious, present information about its functionality, and sometimes provide information to create some uncomplicated network signature. Malware detection here is divided into various phases such as detection, the pre-processing phase, the extraction phase, and the feature-analysis phase [14]. The feature-extraction phase extracts the critical information by parsing the application's source code to form patterns for classifying malicious applications.

Dynamic Analysis: A few checks are typically run during this process, for example, API calls, system calls, network calls, etc. This technique of detection aims to evaluate malware in a natural environment by executing the program [12]. However, antivirus applications rarely perform this type of in-phone analysis as it may not be in the best interest of all parties since scans require significant amounts of memory and CPU power while mobile devices have limits on their computational resources.

Advanced malware analysis tools may employ event generators and observe suspicious behaviour to flag potentially harmful applications. For example, MEGDroid proposed by Hayyan et al. [15] can simulate an environment that triggers malicious payloads to express harmful behaviour. However, this analysis requires an antivirus tool to actually run the suspicious application in a sandbox and therefore cannot be implemented on user devices. This limitation means that antivirus tools rely mostly on feature-based static analysis which, in turn, explains the successful use of obfuscation techniques by malware developers.

1.1.3. Obfuscation Strategies

Malware developers are locked in a constant race to avoid detection from antivirus engines. A popular method for achieving this is obfuscation which intends to modify the executable elements and help the APK to evade detection. Obfuscation is also employed by application developers to ensure security against malware authors and to protect the

application from being reverse engineered. Research has been conducted by various authors on this topic, some of which is reviewed below.

1.2. Related Works in Obfuscated Malware Analysis

Rastogi et al. [16] evaluated the efficiency of anti-malware products for detecting malware subjected to trivial and non-trivial obfuscations. The study proved that 10 out of 10 anti-malware products used for tests failed to detect the applications that had undergone code obfuscation. The outcomes derived from the research on the obfuscation of malware also showed that obfuscating malware can have a disadvantage which states that the malware loses its malicious function, causing no damage to its victim's system. In other words, significant mutations in the virus body may make it less harmful or not harmful at all. Additionally, anti-malware tools such as VirusTotal lack the capability of developing resilience against such obfuscations as they do not always update their signature database after a malicious variant of the application is detected. Nine days elapsed before the anti-malware tools used in their study were able to detect, analyze, and develop signatures, leaving substantial time in which to infect mobile devices. Out of the 10 leading anti-malware providers, only 57% of their signatures provided code-level artifacts. The study revealed that 43% of signature identifications did not focus on code-level artifacts and that component names in the Android manifest were the only way to identify defects. The study also indicated that 90 percent of signatures did not require a static bytecode review since much of the information was contained in the classes—dex file of the application with an Android runtime code [16].

In their study, Hammad et al. [17] discovered that an anti-malware product's detection ability depends both on the obfuscation methodology and the tool used for obfuscation. The detection rate by top anti-malware products dropped by about 20% after malware was obfuscated. The same study proved that applying a specific set of transformations, both trivial or non-trivial, along with the use of commercial obfuscation tools, can achieve a high anti-malware evasion rate, a more extended survival period, and less accurate signature detection [18].

Ajiri et al. [18] also looked at the effectiveness of antivirus engines against obfuscated Android malware. The report looked at five Android malware variants that belonged to ten different malware families before obfuscation, and their detection ratings were used. Then, they were compared with obfuscated Android malware by applying three obfuscation techniques, namely string encryption, renaming, and control flow both individually and in combination. After the individual implementation of obfuscation techniques, their detection ratio decreases significantly, and when a combination of the obfuscation techniques was applied, the likelihood of the detection rate significantly reduced. For example, the research analysis showed that when using a combination of three obfuscation techniques (Control flow, Renaming, String Encryption), an average of only 23.19% samples out of the 50 malware samples (five samples each under ten families) were detected by 60+ analyzers under VirusTotal. Without obfuscation, the average detection rate was much higher: 54.58%. The authors also mentioned that a further step is required for this research study to perform a dynamic analysis of obfuscated Android malware to capture their system calls and compare their results with system calls invoked by non-obfuscated Android malware.

Another study by Malik and Khatter [19] indicated that the detection of obfuscated malware is insufficient with static malware analysis tools and techniques. System call analysis is a powerful technique for malware that is highly encrypted or obfuscated with other methods. Obfuscated malware uses the same system calls (although with different numbers) and performs the same file and network operations during the runtime. To verify the hypothesis, the study focused more on the behavioral characteristics of malware. They used a trace tool for system-calls extraction and extracted 345 Android malicious APKs that belonged to ten Android malware families. In their findings, they discovered that typically malicious applications initiate more system calls than benign apps. The study also identified "suspicious" system calls that were actively used by malware.

Wu et al. [20] proposed contrastive learning to detect obfuscated malware. They tested the approach on 8000+ malware samples achieving a 99% detection ratio and obfuscation resilience. Unfortunately, similar to [17], the study did not indicate the success of their method of obfuscation in terms of if the obfuscated malware was actually functional after obfuscation.

In their work, Sihag et al. [21] looked at the ways developers of legitimate applications and developers of malware can protect their products from reverse engineering or detection. They provided a comprehensive taxonomy of obfuscation techniques that have been proposed in the research literature but the practical evaluation of these techniques and their combinations were not in the scope of the review. That leaves open the question of the applicability of various obfuscation techniques in actual complex applications. Even though these techniques do not change the main logic of the application in theory, their automated implementation with existing tools may actually incapacitate the applications.

All the works reviewed above indicated that obfuscation tools are readily available for application developers and obfuscation techniques indeed significantly reduce the malware detection ratio. Even though some experimental malware detection tools achieve better detection rates they are still not the mainstream solutions used in the industry. Unfortunately, the reviewed research on malware obfuscation did not focus on the question of the intrinsic quality of automated malware obfuscation. In other words, it is not clear if after the automated application of obfuscation techniques advanced applications such as malwares are still functional and harmful.

The research project presented in this paper aims to study the impact of obfuscation on the malware functionality and detection ratio. The impact of obfuscation on malware functionality can be studied through the analysis of the installability and runability of the obfuscated software. In other words, such an analysis would detect whether obfuscated malware can be actually installed on an Android platform and if it can be launched without an immediate crash. This is crucial information because automated tools perform "blind" obfuscations that may incapacitate the malware. This research aims to answer the following questions:

1. Can feature extraction prove helpful in identifying APKs that have been subjected to obfuscation?
2. What is the most effective obfuscation method out of the ones being implemented?
3. Which obfuscation method produces the most installable and runnable APKs?
4. Which obfuscation method produces the most non-installable and runnable APKs?

2. Materials and Methods

The approach implemented in this study uses static and dynamic analyses in a large sample set of original and obfuscated Android applications. The applications included in the sample set comprise malware and benign software. To automate the research steps and minimize human errors, Python script was used to perform automated data gathering and data transfer between various stages of the experiment. A detailed diagram of the research steps and the data flow between them is presented in Figure 2. The description of the steps provided below details all the actions and obtained results.

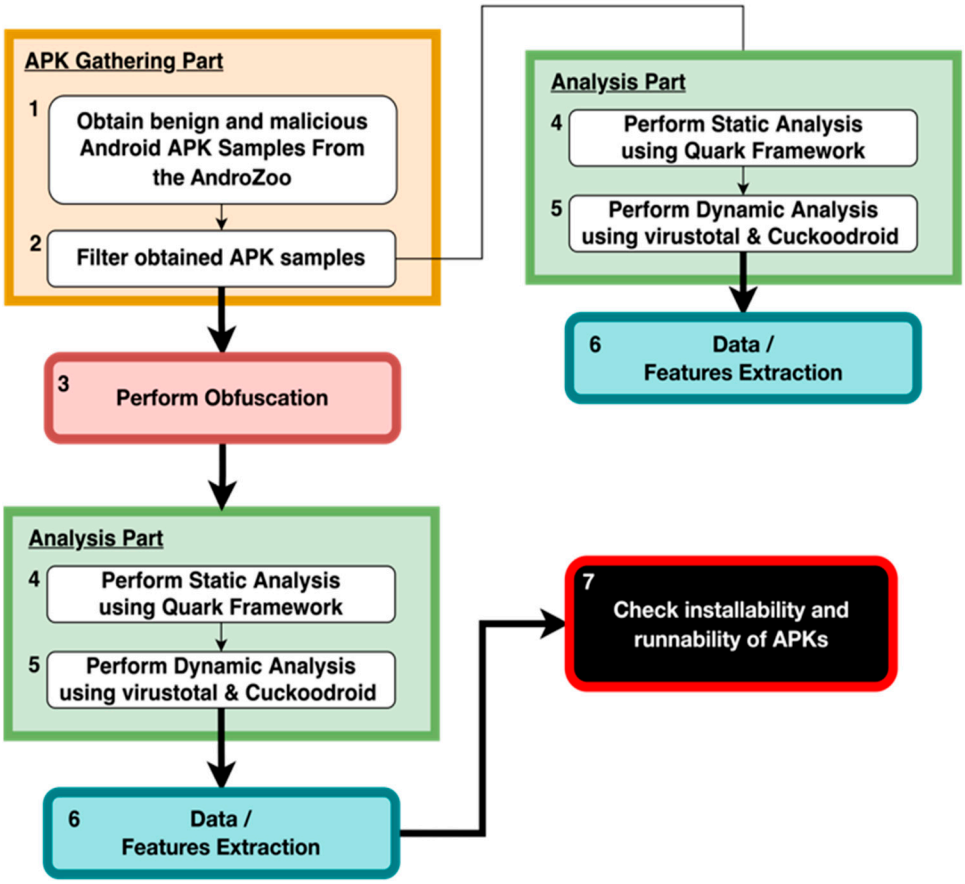

Figure 2. Research Methodology.

2.1. APK Gathering

Android Applications for this experiment were collected from the AndroZoo app database. AndroZoo is a growing collection of pre-analyzed Android apps that are sourced from several sources, including the official Google Play application market [22]. Apart from AndroZoo, the sources of applications included Google Play Store, SlideMe App store, and Anzhi App market. These markets were included for better coverage of applications available for the Android operating system.

For the experiment, 5000 APKs from 2013 to 2016 were selected from AndroZoo. They were filtered by their vt_detection = [0.30+] attribute which indicates the results of detection from Virus Total, where 0 represents benign applications and values above 30 represent applications marked as malware by 30+ anti-virus scanners.

2.2. Obfuscation

The process of performing transformations on an Android application is called obfuscation. These transformations can either be in the form of a single or polymorphic transformation. The Android ecosystem has established a categorization of obfuscation techniques into the following two main groups: trivial and non-trivial [20].

2.2.1. Trivial Techniques

The trivial obfuscation techniques do not change the semantics of the code but can help malware evade signature-based detection by the anti-malware products. In this experiment, the following four trivial techniques were used as presented in Table 1: Align, Re-sign, Rebuild, and Randomize Manifest [5].

Table 1. Trivial Obfuscation Techniques.

Method	Transformations
NewAlignment	Application code is realigned
NewSignature	A new custom signature can be used to re-sign the application
Rebuild	The application is rebuilt using the new obfuscated parameters
RandomManifest	The entries in the manifest file are reordered randomly

2.2.2. Non-Trivial Techniques

In contrast to straightforward trivial techniques, non-trivial techniques may offer a lower detection rate and greater robustness. Resources, including bytecode and other such resources (XMLs, asset files, and external libraries), are the targets of non-trivial obfuscation [5]. In this research, three subcategories of non-trivial obfuscation techniques, namely Renaming, Encryption, and Code [5] were used as shown in Table 2.

Table 2. Non-Trivial Obfuscation Techniques–Rename.

Method	Transformations
ClassRename	Replace the package name and rename classes
FieldRename	Fields are renamed
MethodRename	Methods are renamed

Renaming: Source code, and subsequently Java bytecode, should have meaningful names for identifiers such as variables, functions, and so on to enhance code maintainability. The exact names, however, may expose code functionality. In addition, as the package name uniquely identifies an application, a change to the package name essentially means that the app is being placed into the Android ecosystem as a new application. Thus, each identifier is renamed into an obscure and meaningless one, using the renaming technique presented in Table 2 [5].

Encryption: In an APK file, the developer can specify what resources to request at each run time. For example, a string or a native Android library might be run. With the encryption obfuscation, both the code and resources can be encrypted in application packages and decrypted during the execution phase by applying secret keys of obfuscation tools. Table 3 shows details of the encryption methods used in the experiments [20,21].

Table 3. Non-Trivial Obfuscation Techniques–Encryption.

Method	Transformations
AssetEncryption	Asset files are encrypted
ConstStringEncryption	Constant strings in the overall code are encrypted
LibEncryption	Native libraries are encrypted
ResStringEncryption	Resource strings inside the code are encrypted

Code: Code obfuscation techniques involve modifications to the source code that affect instructions inside the classes.dex file. Several different techniques have been developed to hide the application's behavior, each addressing a different aspect of the code [5]. Techniques used in the experiment are described in Table 4.

Table 4. Non-Trivial Obfuscation Techniques–Code.

Method	Transformations
Reflection	Existing code is examined to find invocations of the main application method while ignoring the Android framework calls. This method can be called using the Reflection APIs.
AdvancedReflection	Using reflection, dangerous APIs from the Android Framework are invoked.
ArithmeticBranch	Uses the junk code insertion technique. A branch instruction is crafted in such a way that the branch is never taken, which results in a piece of junk code composed of arithmetic computations and branch instructions.
CallIndirection	Adds new methods that invoke the original ones. Modifies the control-flow graph without touching the code semantics.
DebugRemoval	The debug meta-data are removed.
GoTo	The software will insert a GoTo into the method and a second GoTo after the first GoTo at the end of the method so that the control-flow graph is modified.
MethodOverload	This exploits the Java overloading feature to return different methods with the same name but with varying arguments.
Nop	Random Nop (No Operations) instructions are inserted into every method implementation.
Reorder	The order of blocks changes in this technique. An inverted condition and reordered basic blocks are created when a branch instruction is found

Table 5 outlines the six different obfuscation strategies implemented for conducting this research. The obfuscation was performed by using the Obfuscapk tool [5].

Table 5. Obfuscation Strategies.

Obfuscation Strategy	Methods
Encryption	AssetEncryption, ConstStringEncryption, LibEncryption, ResStringEncryption
Code	AdvancedReflection, ArithmeticBranch, CallIndirection, DebugRemoval, Goto, MethodOverload, Nop, Reflection, Reorder
Rename	ClassRename, FieldRename, MethodRename
Low	ClassRename, AssetEncryption, AdvancedReflection, MethodOverload, Goto
Medium	ClassRename, FieldRename, ConstStringEncryption, ResStringEncryption, AssetEncryption, AdvancedReflection, MethodOverload, ArithmeticBranch, CallIndirection
High	ClassRename, FieldRename, MethodRename, ConstStringEncryption, ResStringEncryption, AssetEncryption, AssetEncryption, AdvancedReflection, MethodOverload, ArithmeticBranch, CallIndirection, DebugRemoval

2.3. Static Analysis

In the static analysis stage, the application is decompiled to obtain the following four features that are used to classify the application: permissions, native-permissions, intent-priority, and sensitive functions [23]. The Android operating system provides a control mechanism by which to restrict application permissions [24] as a security feature but associative functions can be abused if the application wants to execute a specific function without declaring the appropriate permission in the manifest.xml file. Permissions are used to control applications' functions and to manage the resources of the mobile phone.

Given these relatively mild restrictions, the Android OS can be seen as a relatively open environment and some malware developers may utilize this feature to hide the real purpose of applications or embed malicious functions within normal ones for malicious purposes. Table 6 lists permissions that can be seen as dangerous as they can be misused by application developers to perform functions that are not declared in the application description [24]. For example, the INSTALL_PACKAGES permission may be used by a seemingly benign application to install malware.

Table 6. Dangerous permissions.

Permission Name	Permission Name
ACCESS_BACKGROUND_LOCATION [Added: API level 1]	USE_SIP [Added: API level 9]
ACCESS_COARSE_LOCATION [Added: API level 1]	MODIFY_PHONE_STATE [Added: API level 1]
ACCESS_FINE_LOCATION [Added: API level 1]	WRITE_CALENDAR [Added: API level 1]
CALL_PHONE [Added: API level 1]	INSTALL_PACKAGES [Added: API level 1]
READ_PHONE_STATE [Added: API level 1]	WRITE_CONTACTS [Added: API level 1]
READ_SMS [Added: API level 1]	READ_CALENDAR [Added: API level 1]
RECEIVE_MMS [Added: API level 1]	GET_ACCOUNTS [Added: API level 1]
RECEIVE_SMS [Added: API level 1]	READ CONTACTS [Added: API level 1]
RECEIVE_WAP_PUSH [Added: API level 1]	READ_CALL_LOG [Added: API level 16]
READ_EXTERNAL_STORAGE [Added: API level 16]	WRITE_APN_SETTINGS [Added: API level 1]
ACCESS_MEDIA_LOCATION [Added: API level 29]	RECORD_AUDIO [Added: API level 1]
ACTIVITY_RECOGNITION [Added: API level 29]	CAMERA [Added: API level 1]
ANSWER_PHONE_CALLS [Added: API level 26]	SEND_SMS [Added: API level 1]
BODY_SENSORS [Added: API level 20]	WRITE_CALL_LOG [Added: API level 16]
READ_PHONE_NUMBERS [Added: API level 26]	PROCESS_OUTGOING_CALLS [Added: API level 1, Deprecated: API level 29]

Another area examined with the static analysis includes intent priority analysis. The Manifest.xml file in an Android application defines intent-priority, which identifies the priority of program activities [8]. For example, Application A has a higher intent-priority value than Application B. In that case, related messages will be sent first to A. Most malware raises the intent-priority value to ensure they see information before normal software. Static analysis also examines function calls made by sensitive functions. As part of the static analysis, this study analyzes how often sensitive functions are utilized by an application. Table 6 listed the most common permissions that can be seen as dangerous during a static analysis review.

Manual verification was also used to verify if any parameters (permissions, activities, services) had changed while comparing the original APK to the obfuscated APK. For instance, the manifest files of the original APK and the obfuscated APK were compared to identify if any permissions were added or deleted in the new manifest file of the obfuscated application. Random APKs were selected from the dataset and the comparison was performed between the manifest files of the original APK and the obfuscated version of the same APK.

2.4. Dynamic Analysis

2.4.1. Automatic Dynamic Analysis

For the dynamic analysis using VirusTotal API [25], the original and obfuscated APKs were submitted to VirusTotal and results were retrieved thereafter. The results were based on the execution behavior analyzed by any two of the Android Sandboxes, namely R2DBox and Droidy, used by VirusTotal. The process of submitting and retrieving results was performed with the help of custom Python scripts to facilitate a large number of sample

submissions and analyses. The analysis achieved the following two goals: (1) estimating the detection ratio for the obfuscated malware and (2) evaluating the installability/"survival" ratio of the obfuscated software as automated obfuscation may potentially damage the functionality of the applications. Of course, automated checkups by sandboxes may provide false-positive and false-negative results for the installability of the obfuscated software. To address this potential issue, random sample sets of the obfuscated applications were subjected to manual dynamic verification.

2.4.2. Manual Dynamic Analysis

For manual verification, original and obfuscated APKs were installed and executed in the Android Studio to check if the APKs had survived the different obfuscation methods and had been executed in the same way as the original ones or not. During the manual execution of applications, the following parameters were captured: package names under which apps were running and system calls APKs were calling in the original and obfuscated APKs. For the system calls, the following characteristics were captured: system call name, time percentage for the call, number of times the call was invoked, frequency, and errors [26]. System calls help a malware analyst to understand the behaviour of the application. This data extraction was performed with the help of the Strace tool in the adb (Android debugger) shell. For the manual dynamic analysis, 74 APK samples were randomly selected from the dataset of obfuscated and original APKs.

2.5. Data Extraction

Data extraction was embedded as a part of the static and dynamic analyses. The Quark Framework was used to generate results in the JSON format for static analysis and in the dynamic analysis, Excel spreadsheets were employed to record the results.

2.6. Installability Verification

The original and obfuscated applications were installed on Android emulators [27] to check their installability and verify the number of working applications produced by every obfuscation method. For successful execution and analysis, Anbox and Android Studio were used for loading the applications. Python scripts were constructed based on a methodology that was customized to the specific requirements.

The code continuously works in a loop by downloading the APKs from the AndroZoo using API calls. Upon successfully downloading the APK file, the function "static analysis" was executed. This function uses the Quark Framework, which performs the static analysis and generates the report for a particular APK. A report generated by the function is stored in the folder "Report". After a static analysis of the APK has been completed, the APK can then be imported into an analysis function called "dynamic analysis" that uses the Cuckoodroid [28] to analyze and create a report.

Once the APK File was analyzed both statically and dynamically, it passed through the Obfuscation function, producing six different obfuscated APK files using six different Obfuscation techniques (Rename, Encryption, Code, Low, Medium, High). To accomplish this step, a modular Python tool, Obfuscapk was used. APK files obfuscated by these programs were again submitted for dynamic and static analysis and reporting purposes. In addition, these files were imported into an emulator to verify whether they survived the obfuscation. A Python module was used for the Android bridge driver. Afterward, the user receives a CSV file showing the installed applications and those that were not installed. The source code of the script is available at https://github.com/ddeepp109/Android-Malware-Analysism (accessed on 1 May 2022) [29].

3. Results

3.1. Findings: Obfuscation Strategies

Table 7 shows how different types of obfuscation have a varying impact on the malware detection ratio. The original dataset was obfuscated using Obfuscapk with varying levels

of obfuscation methods as described in Table 5 in the previous section. The outcome of the experiments indicated that the detection rate of VirusTotal for the original set of malicious applications was 91%. This detection rate dropped to 71% on altered apps where the Medium obfuscation strategy was used. It dropped further to 66% on obfuscated apps when the Encryption strategy was used, and to 65% when the High obfuscation strategy was used. It was also observed that the Low obfuscation strategy had less of an impact on malware detection.

Table 7. Detection Ratio based on Obfuscation Strategies.

Obfuscation Technique	Detection Ratio	Percentage
Encryption	3498/5299	66.03%
Code	3602/5299	67.98%
Rename	3815/5299	72.00%
High	3443/5299	64.99%
Medium	3867/5299	72.99%
Low	4132/5299	77.98%

Another noticeable outcome derived in this work was that the impact of trivial and non-trivial obfuscation techniques had almost similar detection rates. A counterintuitive conclusion that can be derived by considering an Android APK is that an archive with a large number of files and a malicious component can be found almost anywhere, and it is not possible to determine which of the above-mentioned techniques should be used if automated obfuscation is applied.

3.2. Findings: Impact of Obfuscation on Static Analysis

A random sample of 2000 applications from the benign and malicious sample sets was selected for the static analysis. Each APK was decompiled using Quark [30] to extract the following five features: (1) permissions requested; (2) native API calls; (3) certain combinations of native API calls; (4) sequence of native API calls; and (5) API calls that handle the same register. In the total 2000 APKs submitted for static analysis, Quark detected all the malware APKs in the original files. However, the detection ratio decreased to 82% for monomorphic obfuscation techniques while producing the lowest detection ratio for High obfuscation of 72%, in which polymorphic obfuscation strategies had been enabled. Table 8 shows the detection ratio of the static analysis for varying levels of obfuscation.

Table 8. Detection Ratio based on Static Analysis.

Obfuscation Method	APKs Tested	APKs Detected	Percentage
Encryption	5299	4371	82.50%
Rename	5299	4398	83%
Code	5299	4191	79.10%
High	5299	3831	72.30%
Medium	5299	3974	75%
Low	5299	4451	84%

The manifest files were compared to determine the impact of obfuscation on the permissions listed in the original APKs. Random APKs were selected from a dataset of 30,000 APKs to check if the permissions were added or deleted from the obfuscated file compared to the original file. Meld [31], a static-analysis tool, was used to perform two-way and three-way comparisons of the files. It was confirmed that all APK's from the sample set have the same permissions in both the original and obfuscated manifest files. However,

in some APKs, the obfuscation impacted the number of times each permission was used. Table 9 provides a sample of such applications.

Table 9. Comparison of permissions in the manifest file of the original APK with the obfuscated APK.

APK	Permissions in the Original APK	Permissions in the Obfuscated Apk
xxxxB917	23	21
xxxxD1C8	5	4
xxxx7C07	15	9
xxxx93FF	10	10
xxxxD60C	10	10

A special feature of Android since API Level 23 is dynamic permission support [32], which allows apps to request, acquire, and revoke permissions as they run. According to this new runtime-permission mechanism, static approaches will not be able to detect abnormal permission requests and grants made during runtime. In addition, users may revoke dangerous permissions after their apps are installed, which could lead to a false positive from anti-malware software.

3.3. Findings: VirusTotal Dynamic Analysis Findings

Table 10 shows the results after the samples were uploaded to VirusTotal and after behavioral reports were fetched from the following two VirusTotal Sandboxes: R2DBox and Droidy. All the results are available on the supplementary website: https://sites.google.com/concordia.ab.ca/evaluation-obfuscated-malware/executability-and-system-calls (accessed on 1 May 2022). The analysis performed using VirusTotal shows that out of all the obfuscations, Medium and High levels of obfuscations had the highest impact on the executability of obfuscated samples, as 70% were found to be showing signs of life–e.g., sandboxes produced some results. In contrast, Low obfuscations showed greater executability as almost 77% of samples produced behavioral results. The single-technique obfuscation strategies—Code, Rename, and Encryption—were shown to exhibit the greatest execution ratio, with 79% of samples producing results. Thus, the ability of obfuscators to produce different variants of a malware sample with fewer detection capabilities and good survival ratios can act as a detrimental tool with which to bypass specific mechanisms deployed for the detection of and protection against suspicious packages. Essentially, it means that malicious actors with medium-level programming skills can produce malware variants at scale with the use of widely available obfuscation tools.

Table 10. Executability of Obfuscated samples seen under VirusTotal Droidy and R2DBox results.

Samples	Obfuscation	Execution Ratio
5299	Encryption	79.39%
5299	Rename	79.36%
5299	Code	79.48%
5299	High	70.48%
5299	Medium	70.04%
5299	Low	77.65%

3.4. Findings: Manual Dynamic Analysis

The goal of this stage was to manually confirm the results obtained at the automated dynamic-analysis stage. The system calls were extracted to compare the behavior of original and obfuscated APKs, as all requests from malicious apps move through the system call interface before being processed.

The APKs that were not executed at all called only nine basic system calls such as read app files, open app files, get process id, get file status, read the clock, and write operations on the files stored in the external storage. The original and malicious APKs that survived the obfuscation actively used process-related functions such as futex, getpid, getuid, gettid, sigprocmask, and prctl. These APKs used sendto() and recvfrom() system calls that are responsible for sending and receiving data from remote servers. Other heavily used system calls that were noticed while performing the manual dynamic analysis were related to accessing data and performing read–write operations on files stored in the external storage and performing memory functions such as read, write, open, close, fcntl64, dup, mmap, munmap, stat64, fstat64, etc. These malicious system calls were used by both original and obfuscated malicious APK samples.

Table 11 indicates the executability of APK samples manually analyzed with the *Strace* tool in *Android Studio*. For detailed results, refer to the supplementary website: https://sites.google.com/concordia.ab.ca/evaluation-obfuscated-malware/executability-and-system-calls (accessed on 1 May 2022). It shows that all obfuscation methods—High, Low, and Medium—impacted the executability of malicious APK samples. The Medium obfuscation method affected the executability the most and decreased it by 15.7%. The following two obfuscation strategies, High and Low, decreased executability by 12.8% and 10%, respectively. Surprisingly, the High strategy had less of an impact on the survival ratio compared to the Medium strategy. This can be explained by the relatively small sample that underwent manual verification. All other methods had a percentage of executability that was almost the same, i.e., near 70%. Moreover, some obfuscated APKs were successfully executed but affected the working of the Android OS. For instance, while analyzing the APK samples manually, it was noted that 7 % of APKs obfuscated using the High strategy, 5% were obfuscated using the Rename strategy, 4% with Encryption, and Medium obfuscated APKs froze or drastically slowed down the emulator. Furthermore, the system calls generated by the original and obfuscation methods were also recorded to identify any changes in their frequency. It was observed that APK samples obfuscated with High and Medium methods generated more system calls in 9% of the APK samples as compared to the original and other obfuscation techniques. The encryption method was intermediate because it generated system calls more often in 7% of the APKs and least in 9% of the APKs as compared to the original and other obfuscation methods. System calls for the remaining majority of the APK samples were almost the same.

Table 11. Executability of APK samples checked with Strace tool in Android Studio.

Sample	APKs	Original	Obfuscation Methods					
			Code	Encryption	High	Low	Medium	Rename
1	14	86%	86%	86%	86%	86%	86%	86%
2	10	100%	100%	80%	80%	80%	80%	80%
3	10	70%	60%	60%	40%	50%	40%	60%
4	10	60%	60%	60%	40%	40%	40%	60%
5	10	60%	70%	80%	40%	70%	60%	80%
6	10	70%	80%	70%	40%	50%	30%	70%
7	10	50%	60%	70%	30%	50%	50%	60%
AVG	74	70.8%	73.7%	72.3%	58%	60.8%	55.1%	70.8%

Summarizing the above, it can be stated that the manual dynamic analysis confirmed a high survival ratio of malware after non-specific obfuscation. In turn, it confirms that malicious actors with even medium-level programming skills can use obfuscators to produce Android malware variants.

3.5. Finding: Application Installation and Runnability

To check the "survival" ratio of applications subjected to obfuscation, they were tested for installability and runnability. To verify the installability of the applications, seven sample sets of applications were prepared. Sample 1 consisted of 14 randomly selected applications and Samples 2 to 7 included 10 randomly selected applications. Samples were relatively small due to the following two factors: (a) manual verification of the results and (b) the need to contain potential errors in installation to a smaller population of applications. For all the samples, the selection included benign and malicious applications.

Each sample set was subjected to six of the various obfuscation strategies described earlier. Samples from the Sample 1 set that were obfuscated were first installed in Anbox Application Manager using the automated script. Out of the 14 applications that were randomly selected for every obfuscation method, 12 applications were successfully installed every time. The other two applications could not be installed. However, these 12 applications were not runnable on Anbox. The applications froze Anbox every time they were loaded into the emulator. Additionally, these applications were manually installed in *Android Studio*. All 14 applications were successfully installed and were runnable.

From samples 2–7, the obfuscated applications did not successfully install in the Anbox application manager. The reason for this may be the higher API level that Anbox runs on. This reason was identified because the obfuscated applications were also installed on *Android Studio* with a higher- and a lower-level API machine. The applications were successfully installed on the virtual Android device with a lower API level of 22 and were not installed on the Android device with a higher API level of 24. It is also worthy of note that the original applications were successfully installed in Anbox and Android Studio. It was clear that blind obfuscation may not always work, and potential attackers would need to pay attention to the API level/version when they prepare obfuscated malware. Table 12 provides detailed information on the results of the installability tests for all seven sets of applications.

The review of data obtained from the installability analysis shows that the obfuscation strategy "Code" produced the highest number of valid applications post obfuscation with 73.71% valid applications. The "Encryption" method produced 72.28% valid applications. The "Rename" method produced 70.85% valid applications. The "Low" method produced 60.85% valid applications. The "Medium" method produced 55.14% valid applications, and the "High" method produced the lowest number of valid applications by producing just 50.85% valid applications.

These results clearly indicate that trivial obfuscation strategies provide better chances for automated obfuscation while the automated application of more sophisticated/polymorphic obfuscation strategies may produce applications that are not actually functional. Figure 3 compares the results obtained from the static and dynamic analyses along with an installability check. It was observed that the application of multiple levels of polymorphic obfuscation bypassed both the static and dynamic detection algorithms and at the same time caused a greater degree of changes in the application semantics resulting in non-functional malware APK. On the other hand, trivial and monomorphic obfuscation produced APKs with a higher detection ratio but maintained the semantics of the APKs. Overall, the Code obfuscation strategy produced the most optimum results with a lower detection ratio and higher installation probability.

Table 12. Installability of Obfuscated Applications.

APK	Original	Obfuscation Methods					
		Code	Encryption	High	Low	Medium	Rename
Sample 1				14 × 7 APK			
Anbox	100%	86%	86%	86%	86%	86%	86%
Android Studio	86%	86%	86%	86%	86%	86%	86%
VirusTotal S1	0%	93%	93%	93%	93%	93%	93%
VirusTotal S2	0%	50%	64%	71%	57%	57%	64%
Sample 2				10 × 7 APK			
Anbox	100%	0%	0%	0%	0%	0%	0%
Android Studio	100%	100%	80%	80%	80%	80%	80%
VirusTotal S1	0%	70%	50%	70%	50%	80%	70%
VirusTotal S2	80%	70%	70%	60%	60%	70%	60%
Sample 3				10 × 7 APK			
Anbox	70%	0%	0%	0%	0%	0%	0%
Android Studio	70%	60%	60%	40%	50%	40%	60%
VirusTotal S1	0%	40%	40%	20%	30%	30%	40%
VirusTotal S2	60%	50%	40%	30%	50%	40%	40%
Sample 4				10 × 7 APK			
Anbox	60%	0%	0%	0%	0%	0%	0%
Android Studio	60%	60%	60%	40%	40%	40%	60%
VirusTotal S1	0%	60%	60%	40%	60%	30%	60%
VirusTotal S2	100%	70%	60%	50%	60%	40%	60%
Sample 5				10 × 7 APK			
Anbox	60%	0%	0%	0%	0%	0%	0%
Android Studio	60%	70%	80%	40%	70%	60%	80%
VirusTotal S1	0%	40%	50%	20%	30%	30%	40%
VirusTotal S2	70%	50%	50%	10%	20%	40%	60%
Sample 6				10 × 7 APK			
Anbox	70%	0%	0%	0%	0%	0%	0%
Android Studio	70%	80%	70%	40%	50%	30%	70%
VirusTotal S1	0%	40%	30%	30%	40%	20%	30%
VirusTotal S2	60%	50%	50%	30%	60%	30%	40%
Sample 7				10 × 7 APK			
Anbox	60%	0%	0%	0%	0%	0%	0%
Android Studio	50%	60%	70%	30%	50%	50%	60%
VirusTotal S1	0%	40%	40%	20%	30%	30%	40%
VirusTotal S2	60%	50%	40%	40%	60%	40%	50%

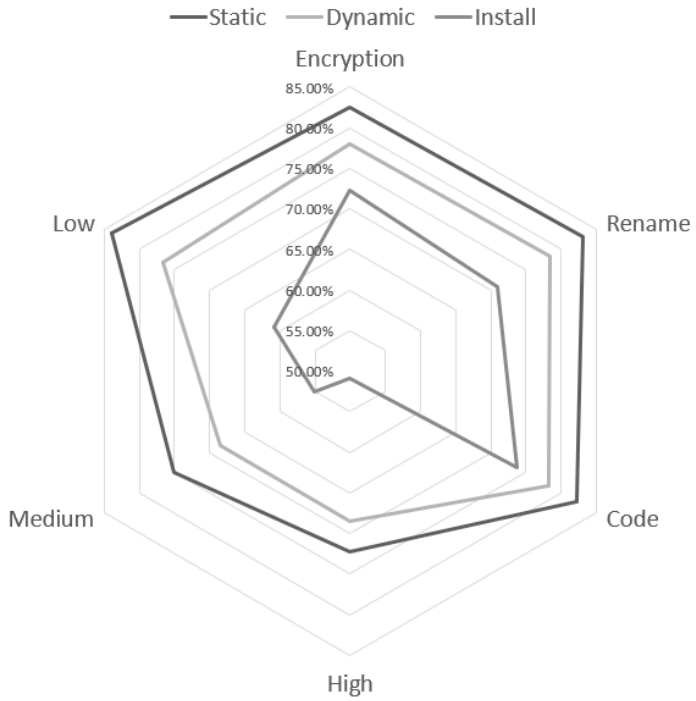

Figure 3. Comparative analysis of Static Analysis, Dynamic Analysis, and Executability of Obfuscated APKs.

4. Discussion

This paper evaluated the effectiveness of static and dynamic analyses against code obfuscation and the survival ratio of malware after varying levels of obfuscation. In the analysis presented above, it was observed that polymorphic obfuscation techniques had a lower detection ratio as compared to monomorphic obfuscation techniques but, on another hand, an automated application of advanced obfuscation may make applications inoperational.

Key findings of the study include the following:

(1) Regardless of the technique applied, of either a dynamic or static analysis, automated obfuscation leads to a decreased detection ratio of malware by major anti-malware products.

(2) The results obtained from the static analysis such as permissions and native API calls produce information that can help to detect obfuscated code. Such information can be further used to deploy an advanced analysis such as that described in [15].

(3) In most cases, a trivial transformation, such as modifying the Android manifest file or rebuilding applications with a new signature, was effective in bypassing detection techniques. This, in turn, indicates that most of the current anti-virus products for the Android platform do not employ advanced detection techniques.

(4) Despite the relatively weak functionality, dynamic system calls when combined with other features extracted through a manual analysis produce effective results, thereby increasing the malware detection ratio.

(5) The APKs' executability was negatively impacted by automatically applied polymorphic obfuscation strategies but most of the applications remained installable after such obfuscations.

(6) While monomorphic obfuscation techniques exhibit strong detection resilience, a mixture of obfuscation techniques, automated polymorphic obfuscation, exhibits an even higher level of detection resilience.

(7) Out of all the obfuscation strategies, the automated application of Code obfuscation proved to be most effective with a lower detection ratio and higher installation probability.

The experimental setup and results obtained in the paper show that there is a need for an improvement in Android malware detection tools with, potentially, more of an emphasis on dynamic analysis. Ease of access to automated obfuscation tools and techniques can be

leveraged by sophisticated attackers or script kiddies to execute a successful malware tool that remains undetected in cases of a targeted attack campaign.

This paper presents data and features generated by the static and dynamic analysis methods, which can be used for future work for a deeper study of how these features can be used to improve the performance of machine learning algorithms for malware detection purposes.

Author Contributions: Software: H.P., D.P., H.S., H.K.; Investigation: J.A., V.K., K.N., G.H.R., S.S.P.; Conceptualization: S.B.; Writing—Original draft: D.P., H.P.; Writing—review and editing: all the participants; supervision: S.B. All authors have read and agreed to the published version of the manuscript.

Funding: The publication of this paper is supported by the Internal Research Grant provided by Concordia University of Edmonton.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Automation script is available on GitHub: https://github.com/ddeepp109/Android-Malware-Analysis (accessed on 1 May 2022). Raw data from the experiments are available at https://sites.google.com/concordia.ab.ca/evaluation-obfuscated-malware/executability-and-system-calls (accessed on 1 May 2022).

Acknowledgments: The authors would like to acknowledge support from VIRUSTOTAL (https://virustotal.com accessed on 1 May 2022) for providing access to its API. The authors would like to acknowledge support from Androzoo (https://androzoo.uni.lu/ accessed on 1 May 2022) for providing access to benign and malicious Android applications.

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

References

1. Statcounter Global Stats. Mobile Operating System Market Share Worldwide. May 2021. Available online: https://gs.statcounter.com/os-market-share/mobile/worldwide (accessed on 12 June 2021).
2. Behera, C.K.; Bhaskari, D.L. Different obfuscation techniques for code protection. *Procedia Comput. Sci.* **2015**, *70*, 757–763. [CrossRef]
3. Collberg, C.; Myles, G.R.; Huntwork, A. *Sandmark—A Tool for Software Protection Research*; IEEE Security & Privacy: Piscataway, NJ, USA, 2003; Volume 1, pp. 40–49.
4. Berzinskas, L. Obfuscating Android Apps: Do You Know Your Choices for Protection? 25 January 2020. Available online: https://proandroiddev.com/obfuscation-is-important-do-you-know-your-options-30b3ef396dfe (accessed on 13 May 2021).
5. Aonzo, S.; Georgiu, G.C.; Verderame, L.; Merlo, A. Obfuscapk: An open-source black-box obfuscation tool for Android apps. *SoftwareX* **2020**, *11*, 100403. [CrossRef]
6. Google. Android Architecture Components. 24 February 2021. Available online: https://developer.android.com/topic/libraries/architecture (accessed on 13 May 2021).
7. Android Developers. Application Fundamentals. 23 February 2021. Available online: https://developer.android.com/guide/components/fundamentals (accessed on 14 May 2021).
8. Google. App Manifest Overview. 20 April 2021. Available online: https://developer.android.com/guide/topics/manifest/manifest-intro (accessed on 20 May 2021).
9. Android Developers. App Manifest Overview. 20 February 2021. Available online: https://developer.android.com/guide/topics/manifest/manifest-intro#components (accessed on 20 May 2021).
10. Kotipall, S.R.; Imran, M. Understanding the app's attack surface. In *Hacking Android*; Packt: Birmingham, UK, 2016.
11. Codemagic. Android Code Signing. 15 June 2021. Available online: https://docs.codemagic.io/code-signing/android-code-signing/ (accessed on 23 May 2021).
12. Or-Meir, O.; Nissim, N.; Elovici, Y.; Rokach, L. Dynamic Malware Analysis in the Modern Era—A State of the Art Survey. *ACM Comput. Surv.* **2019**, *52*, 1–48. [CrossRef]
13. Yusirwan, S.; Prayudi, Y.; Riadi, I. Implementation of malware analysis using static and dynamic analysis method. *Int. J. Comput. Appl.* **2015**, *117*, 11–15. [CrossRef]
14. Bakour, K.; Unver, H.M.; Ghanem, R. The Android Malware Static Analysis: Techniques, Limitations, and Open Challenges. In Proceedings of the 3rd International Conference on Computer Science and Engineering (UBMK), Sarajevo, Bosnia and Herzegovina, 20–23 September 2018.

15. Hasan, H.; Ladani, B.T.; Zamani, B. MEGDroid: A model-driven event generation framework for dynamic android malware analysis. *Inf. Softw. Technol.* **2021**, *135*, 106569. [CrossRef]
16. Rastogi, V.; Chen, Y.; Xuxian, J. Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks. *IEEE Trans. Inf. Forensics Secur.* **2014**, *9*, 99–108. [CrossRef]
17. Hammad, M.; Garcia, J.; Malek, S. 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, New York, NY, USA, 27 May–3 June 2018; pp. 421–431.
18. Ajiri, V.; Butakov, S.; Zavarsky, P. Detection Efficiency of Static Analysers against obfuscated Android Malware. In Proceedings of the IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), Baltimore, MD, USA, 25–27 May 2020.
19. Malik, S.; Khattar, K. System Call Analysis of Android Malware Families. *Indian J. Sci. Technol.* **2016**, *9*, 1–13. [CrossRef]
20. Wu, Y.; Dou, S.; Zou, D.; Yang, W.; Qiang, W.; Jin, H. Obfuscation-resilient Android Malware Analysis Based on Contrastive Learning. *arXiv* **2021**, arXiv:2107.03799.
21. Sihag, V.; Vardhan, M.; Singh, P. A survey of android application and malware hardening. *Comput. Sci. Rev.* **2021**, *39*, 100365. [CrossRef]
22. Allix, K.; Bissyandé, T.F.; Klein, J.; Traon, Y.L. AndroZoo: Collecting Millions of Android Apps for the Research Community. In Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), Austin, TX, USA, 14–22 May 2016.
23. Tchakounte, F.; Dayang, P. System Calls Analysis of Malwares on Android. *Maejo Int. J. Sci. Technol.* **2013**, *2*, 669–674.
24. Google. Manifest.Permission. 9 June 2021. Available online: https://developer.android.com/reference/android/Manifest.permission (accessed on 13 June 2021).
25. Virus, T. VirusTotal API Version 3 Overview. Available online: https://developers.virustotal.com/v3.0/reference#overview (accessed on 22 March 2021).
26. Yuan, H.; Tang, Y.; Sun, W.; Liu, L. A detection method for android application security based on TF-IDF and machine learning. *PLoS ONE* **2020**, *15*, e0238694. [CrossRef] [PubMed]
27. Google Devloper. Run Apps on the Android Emulator. Available online: https://developer.android.com/studio/run/emulator (accessed on 17 June 2021).
28. Idanr. CuckooDroid—Automated Android Malware Analysis. 25 July 2017. Available online: https://github.com/idanr1986/cuckoo-droid (accessed on 22 March 2021).
29. Patel, D. Evaluation of obfuscated android malware. Available online: https://github.com/ddeepp109/Android-Malware-Analysis (accessed on 17 June 2021).
30. Quark-Engine. Available online: https://github.com/quark-engine/quark-engine (accessed on 24 March 2021).
31. Willadsen, K. Meld. Available online: https://meldmerge.org/ (accessed on 12 January 2021).
32. Google. *Request App Permissions*. Google. Available online: https://developer.android.com/training/permissions/requesting (accessed on 19 February 2020).