Vol. 2, No. 1 | January – June 2018

Evaluation of Android Malware Detectors

Hassan Rafiq[1], Muhammad Aleem[1], Muhammad Arshad Islam[1]

Abstract:

Malware is an umbrella term used for viruses, worms, and Trojans. These days malware is becoming a great threat to the Android users. A malware detector which is commonly known as an antivirus or virus scanner avoids a malicious file to infiltrate into a system. With the increasing usage of smartphones, malware is also becoming powerful to penetrate the mobile devices. Traditional protection systems identify malware using signatures that can be manipulated by various techniques. In this research paper, it has been demonstrated that the most of the known commercial malware detectors cannot detect common code obfuscation techniques. Moreover, we have evaluated resource utilization (CPU, memory, and battery) consumed by several malware detectors.

Keywords: *Malware detectors; Antivirus; Android; Hardware performance counter.*

1. Introduction

A malware can penetrate into the host devices through the several ways. For example, a malware can integrate itself with a downloaded file downloaded, or via infected flash drives, or someone can intentionally insert a malicious file into a system. A malware developer can spread a malicious file via email or by attaching it to an application which apparently seems to be legitimate. Generally, malware can be classified on the basis of the propagation methods as discussed by McGraw et al. [1].

A malware can cause a severe damage to the infected devices, for example, it can compromise *confidentiality*, *integrity*, and *availability* of a system or network. Similarly, keylogger class malware can penetrate into a system to steal passwords and other sensitive information. Additionally, a particular type of malware commonly known as ransomware [2] encrypts the data and demand money for the data to be decrypted. Thus a malware can cause loss of important data and also cause huge financial loss to organizations and individuals.

Given the widespread emergence of Android malware, there is a crucial need to adequately moderate or protect against these threats. As indicated by the Intel Security/McAfee April 2017[2] patterns report; towards the end of the year2016, there were more than 600 million malware variations altogether. There were approximately 15 million distinctive portable malware variations by the end of the year2016. According to this report, nearly 08% of mobile users have been infected by some kind of smartphone-based malware. Thus, without an in-depth understanding of mobile malware, it is impractical to develop a reliable solution for the detection of mobile malware. In contrast to the existing mobile operating systems, Android is targeted mostly due to the open-source availability of this operating system [3].

A malware detector or antivirus identifies and scans a file using various mechanisms and checks whether the file is infected (malicious) or benign [4]. Generally, a malware detector executes in a passive mode (in the background) and scans a suspicious file. An

[1] Capital University of Science and Technology, Islamabad
Corresponding Email: aleem@cust.edu.pk

antivirus scans a file whenever a file is accessed or it performs a complete system scan on an explicit user request to scan every file in the system. Generally, a full system scan is applied and helpful when a user has installed an antivirus program (first time) and wants to ensure that there are no malicious programs in a system.

Similar to the personal computers, traditional approaches have been adopted to protect mobile devices too from malware threat. Mostly, malware detectors rely on the virus definitions to detect malware. These virus definitions are updated regularly i.e., every day or more often. Virus definitions mostly consist of signatures of the known malware families and variants. Malware detectors have to continually keep up-to-date with the latest malware definitions to be effective for malware detection. Antivirus tools employ a variety of tools to disassemble malware for analysis. Malware detectors also employ heuristics [4] which make a malware detector more capable to identify new malware even without the up-to-date virus definitions.

In this paper, we have highlighted a potential problem that the most of the commercial malware detectors are unable to detect obfuscated malware samples. With code obfuscation, a developer can hide the original algorithm or the logic of the code [3]. We have experimented using various types of code obfuscation techniques (as listed in Table 4) to benchmark which malware detectors are still able to identify a malicious code obfuscated within a legitimate application. Additionally, one of the key aspects of mobile devices is energy conservation. Therefore, the malware detectors are evaluated on the basis of resource consumption reported by the key performance counters including battery consumption. Our research aims are to benchmark the effectiveness of malware detectors against the obfuscated malware. Following are some of the contributions of this work:

- Using several types of code obfuscation techniques to test Android malware detectors;
- Benchmarking android malware detectors based on their malware detection capability;
- Profiling and analysis of Android malware detectors based on resource usages such as CPU, memory, and energy.

The structure of the rest of the paper is as follows. Section 2 discusses the related previous research works. In Section 3, we present the proposed methodology for benchmarking Android malware detectors. Section 4 presents the experimental results and discussion. Section 5 concludes the paper.

2. Background and Related Work

Android applications are developed in Java. Java source code is packaged into an Application (Apk) file which executes on the Android devices [5].We use dex2jar [6] and apktool [7] to convert the android applications into the source code. After de-compilation into code and resource files, the Apks can be analyzed. In this paper, we de-compile known malware samples and make changes to their code without modifying the applications' functionality.

2.1. Code Obfuscation Techniques

Code obfuscation [8] is mainly done to hide the logic of the code so that the code could not be understood after reverse engineering. Code obfuscation changes the size and content of the Apk file; however, the main logic of the code is not modified. Code obfuscation does not have any impact on the semantics of a code. There are many code obfuscation techniques which can be applied to generate various code versions with the same semantics.

In one of the recent work, Zheng et al. [9] evaluated malware detection capabilities of malware detectors by applying code transformations. The authors developed

different test cases of malware samples by using several transformations and then evaluated using virus total [10] platform.

Authors employ artificial code diversity [11] as a code obfuscation method to evaluate the malware detection platform i.e., virus total. The authors prepared malware samples using a tool named *ADAM*. This tool was developed by the authors and employed for the code obfuscation. As compared to this work, we manually applied several obfuscation techniques after reverse engineering the malicious Apk file. Moreover, we perform testing on well-known commercial malware detectors.

Christodorescu and Jha [12] tested desktop malware detectors in the similar manner as we perform in this study. The results of their experiments show that the most of the malware detectors are unable to detect malware samples. Moreover, we experiment using six malware detectors as compared to three tested by the authors in [12].

Collberg et al. [8] have discussed different kinds of code obfuscation techniques. They presented working and architecture of Java code obfuscating tool named as *Kava*. We use some of the mentioned code obfuscation techniques presented by the authors in [8].

Christodorescu et al. [13] have proposed a technique that suggests that the obfuscated malware samples can be detected. However, this detection is limited to detection of only garbage and re-ordered code. In this work, we use six code obfuscation techniques and their combinations as compared to only three employed by the authors to benchmarks malware detectors.

Protsenko et al. [14] have proposed a tool named as *Pandora* using can be used to apply obfuscation. After that, the obfuscated code can be tested using a malware detection tool such virus total. In contrast, we perform benchmarking of malware detectors using commonly used code obfuscation techniques and six most used malware detectors. In Table 1, a brief summary of the related work is shown.

TABLE 1. Related work summary.

Reference and Methodology	Strengths	Weaknesses
- Semantics persevering obfuscation techniques are applied.	-Obfuscated samples bypass malware detectors.	-Only three malware detectors are tested.
- [6], Different levels of obfuscation are used. -Each level consists of different combinations of code obfuscation techniques.	-Checks software plagiarism based on the proposed technique	-Testing of malware samples is performed on virus total only.
- [10], Variants of a single malware sample are prepared -Each sample is tested using malware detector “virus total”	-Malware samples are automatically prepared using a tool ADAM.	-Testing of malware samples is performed on virus total only.
-[14], Proposed a semantic-aware malware detection technique.	-Can detect a malware sample in which code obfuscation is applied	-Can detect malware based on only garbage insertion, code reordering, and register renaming based
-Proposed a mechanism to detect malicious files -Detects malicious files based on their behavior on the network.	-Obfuscated malware samples can also be detected	-Only applicable for malware which access network excessively

3. Methodology

The detection capability of malware detectors is tested by using obfuscated malware samples which are prepared by performing several different steps as shown in Figure 1. Only those malware samples are takes for code obfuscation which are detected as malware in the original form. (i.e. Before applying code obfuscation).

We prepare six different malware samples from a single malware by applying different code obfuscation techniques. The employed six code obfuscation techniques are listed below:

1. Variable Renaming [11]
2. Package Renaming [13]
3. Method Renaming [9]
4. Garbage Insertion [6]
5. Rebuilding [14]
6. Call Indirection [13]

1) ***Variable Renaming***: All the variable names are modified in the context of variable renaming. Figure 1 shows an example code obfuscation using variable renaming.

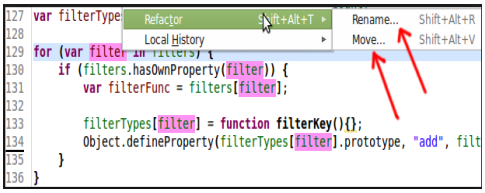

Fig. 1. Variable renaming.

2) ***Package Renaming***: is related to changing package names of a given apk using the *Android Manifest* file.

3) ***Method Renaming***: similarly, in method renaming names of all the method is changed.

4) ***Garbage Insertion***: Whereas in garbage insertion, a garbage code is inserted that does not change the semantics of the application.

Listing 1: Indirect function call and garbage code insertion.

```
1 void display()
2 {
3   cout<<"hello world";
4 }
5 void show()
6 {
7   display();
8 }
9
10 void main( )
11 {
12  display(); //Direct call
      to display
13  //function
14  Show(); //Indirect call
      to display
15  //function
16  while(0)
17  {
18    cout<<"garbage";
    //Garbage code
19  }
```

In Listing 1, a *while loop* is shown with a false condition. The execution control never enters such loop and the enclosed code will not be executed. Such kind of code is referred as garbage code and can be inserted by the malware writers to create code level dis-similarity in malware applications.

5) ***Rebuilding***: Another effective technique that can be used to test the malware detection capability of an antivirus or malware detector is *code rebuilding*. When rebuilding a code, the Apk is first decompiled and then is recompiled without making any changes in its resources and manifest file. Rebuilding process does not change the content of the Apk; however, it generally changes the byte order [9] and the hash value of the application. In most of the malware detectors, the detection algorithms mainly rely on the hash signatures of the files under investigation. Therefore, malware writers exploit this fact to doge the malware analysis tools.

6) ***Call indirection:*** is another effective technique in which the original method calls are re-programmed and shifted in some dummy methods to make indirect function

calls. Listing 1 at line 14 shows an example of *call indirection.*

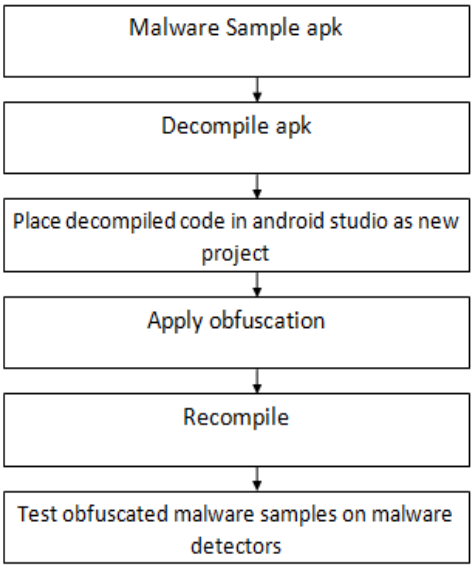

Fig. 2. Proposed methodology.

Figure 2 shows the proposed methodology used to benchmark malware detectors. The first step shows that a sample malware Apk is taken. We use malware application dataset available at [15]. The malware sample may belong to any known malware family. We choose only those malware samples which could be detected by the employed six malware detectors.

The second step of the methodology is based on decompiling Apks using dex2jar [6] and apktool [7] into Java code. In the third step, a new Android project is created based on the decompiled Java code and XML design files.

In the fourth step, obfuscation is applied to the decompiled code. To obfuscate the code, we employ the six obfuscation techniques and their combinations. The output of each obfuscated method is a new version of the Apk; for example, after changing names of all the variables the code is recompiled the version of the Apk is saved separately. To insert garbage-code, a redundant non-executable code is inserted (as shown in Listing 1) and is recompiled to generate the Apk. Similarly, method calls indirection is used to invoke methods via some other indirect method as shown in lines 5-8 in Listing 1. In addition to the six obfuscation techniques, several other combinations (shown in Table 4) are used to generate several versions of the Apks.

4. Results and Discussion

4.1. Dataset and Experimental Setup

The experimentations were performed using an Android system with following specifications, i.e., CPU 1.3GHZ, quad-core, 01GBs of main memory, battery 200 mAH and Android version 4.2 (jelly beans). Table 2 shows the names of the Android malware detectors which have been tested.

TABLE 2. Malware detectors evaluated.

Product name	Total downloads (millions)
Norton Mobile Security	5M-10M
AntiVirus Free	50M-100M
ESET mobile security	500K-1M
Dr Web	10M-50M
Lookout mobile security	10M-50M
Zoner Antivirus	1M-5M

TABLE 3. Malware samples used for testing.

Malware	Details
Love Trap	A trojan that sends SMS
DroidDream	Creates spoof version of the original application
FakePlayer	Advertises unwanted products
Bgserv	Fake mobile cleanup tool
Basebridge	Performs harmful actions without user's knowledge
Plankton	Sends host's information to a remote server
Geinim-A	Corrupts the applications
LuckyCat	Opens backdoor in application to steal information
HippoSMS	Sends SMS to a hard-coded number
NickySpy	Sends host's information to are mote server

Table 3 shows the names and functionality of the malware samples which are used to prepare the test cases to evaluate the malware detectors shown in Table 2.

All the malware samples shown in Table 3 are detected as malicious in their original form (i.e., before obfuscation is applied) by all the malware detectors shown in Table 2. Some of the malware detectors have been omitted from Table 2 because they were unable to detect the malware samples as malicious which are shown in Table 3. All the malware detectors are directly downloaded from the official Android application market i.e., Google Play. Table 4 shows the list of obfuscation techniques that have been used in this paper to evaluate malware detectors.

TABLE 4. Labels of code obfuscation techniques.

Labels	Technique
VR	Variable Renaming
MR	Method Renaming
REB	Rebuilding
GCI	Garbage code insertion
RP	Package renaming
CI	Call indirection

4.2. Results

Table 5 shows the minimal combinations of obfuscation techniques required to evade a malware detector. For example, *LoveTrap* requires variable renaming, method renaming, and package renaming to evade Norton antivirus, Antivirus free, ESET and Lookout. Love Trap remains undetected by Dr. Web if package renaming and call indirection is applied, whereas Zoner cannot detect LoveTrap if simple rebuilding is applied to it.

Similarly, when we consider DroidDream malware sample then the results shown in Table 5highlight that the Norton and the ESET cannot detect DroidDream sample for the combination of package renaming and rebuilding obfuscations. Whereas, in case of Dr. Web malware detector, the *Lookout*,*Zoner*, and the *DroidDream* samples go undetected (with simple application re-building obfuscations).

In case of Bgserv malware sample, the results of Table 5 show that the Norton malware detector is evaded by the obfuscation combination of package renaming, variable renaming, and method renaming. The AntiVirus free is evaded by the obfuscation combination of package renaming and call indirection. The malware sample Bgserv could not be detected as malicious by the ESET and *Lookout*tools for the obfuscation combinations based on package, variable, and method renaming. The malware detector Dr. Web also could not detect *Bgserv* malware sample based on call indirection obfuscation. The malware detector Zoner could not detect Bgserv as malicious even when a simple rebuilding was applied to it. The Hippo SMS malware evaded malware detection capability of *Antivirus Free*, *ESET*, and *Dr. Web* when a combination of package renaming and rebuilding was applied. The hippo SMS evaded Lookout and Zoner malware detectors when the malware sample was simply rebuilt.

Keeping in view the results of Table 5, we may conclude that the Norton antivirus is a hard nut to crack because it can only be evaded if complex obfuscation is applied to a malware sample i.e., a combination of variable renaming, method renaming, and package renaming. On the other hand, the Zoner malware detector proves to be the weakest among the employed anti-malware because it can be evaded by simply re-building a malware sample. The rest of the malware detectors (as shown in Table 5) are not resilient to several combinations of code obfuscation techniques. Most of the malware detectors are able to detect the re-build samples; however, they are unable to detect malware samples when several obfuscations are used collectively.

Figure 3 shows the malware detection results for different malware detectors against the employed obfuscation techniques. In Figure 3 , the Y-axis shows the tested malware detectors and X-axis shows the percentage of

samples evaded the employed malware detectors. Figure 3 presents the results of the malware sample *Hippo SMS* and its versions based on code obfuscation. For Norton antivirus, the results show that most of the code obfuscations have been detected; however, the combination of *Package Renaming* (RP), *Variable Renaming* (VR), and *Method Renaming* (MR) obfuscation techniques resulted in 70% undetected cases. For the combination of *package renaming* and *rebuilding* results in only 20% of un-detected cases for the Norton antivirus. In our experiments, we observed that the *variable renaming*, *method renaming*, package *renaming*, *call* indirection, and simple rebuilding are easily detectable using the Norton antivirus. Moreover, Figure 3 shows the detection results of other antiviruses for the employed code obfuscation methods. As shown in Figure 3, simple code rebuilding is detected by most of the antiviruses except *Dr web* (30% samples undetected) and *lookout* (10% samples undetected). The combined obfuscation based on *package renaming* and *call indirection* also show a large percentage of un-detectable malware samples. The results show that the most stealth obfuscation samples were based on the combination of *package*, *method*, and *variable* renaming. Similarly, a higher evading result was shown for the code obfuscations based on simple package renaming combined with call indirect.

Next, we perform resource consumption analysis for the employed 06 android malware detectors. Table 6 shows the results obtained using the benchmarking tool Mobibench [16]. Table 6 presents the resource consumption chart for 06 malware considering the CPU, memory, battery, and storage requirements. Mobibench employs Android APIs to calculate memory and processor usage. To calculate battery consumed by a malware detector, the Mobibench requires an Android device to run in a *clean state* (i.e., no other application being executed at that time of instance). Mobibench records the battery level of the device before starting the malware detector and again record the battery level after the malware detector finishes its task (of screening). The battery consumed is shown in units milli-ampere-hour (mAH) as shown in Table 6.

Table 6 shows that *Dr Web* consumes 16% CPU, 56% RAM or memory, 0.91 mAH battery, and 7.13 MBs size on disk. Similarly, the performance analysis of other malware detectors is shown in Table 6. These results show that the Norton antivirus is the highest resource consuming malware detector whereas the zoner malware detector consumes the least device resources as compared to other malware detectors.

5. Conclusion

The experiments performed in this research show that there are serious shortcomings in the commercially available malware detectors (against the obfuscated malware). To demonstrate these, we employ several malware detectors and tested those using many combinations of code obfuscations. Most of the time, an obfuscated malware is undetectable. However, a few Android malware detectors (such as Norton, antivirus free, etc.) are able to detect malware obfuscated using multiple techniques. The results clearly show that well known commercial malware detectors are not resilient to common code obfuscation techniques. In addition to this, it has also been observed that the malware detectors which have good detection rate also consume more device resources especially battery and storage space. In future, we intend to research the mechanism using which a malware detector should be able to detect the obfuscation applied to the original malware sample; hence, improving overall malware detection rate.

TABLE 5. Evaluation summary.

	Love Trap	Droid Dream	Fake Player	Bgserv	Base Bride	Plankton	Geinim-A	Lucky Cat	HippoSMS	NickySpy.B
Norton	RP+VR+MR	RP+REB	RP+VR	RP+VR+MR	VR+MR	RP+VR+MR	RP+VR+MR	VR+MR	RP+VR+MR	RP+VR+MR
Antivirus Free	RP+VR+MR	VR+MR	RP+VR+MR	RP+CI	RP+CI	RP+VR+MR	VR+MR	RP+REB	RP+CI	RP+VR+MR
ESET	RP+VR+MR	RP+REB	CI	RP+VR+MR	RP+VR+MR	RP+CI	RP+REB	RP+REB	RP+CI	RP+CI
Dr. Web	RP+CI	REB	RP+REB	CI	CI	CI	RP+REB	RP+REB	RP+CI	RP+REB
Lookout	RP+VR+MR	REB	RP+VR	RP+REB	RP+CI	REB	REB	REB	REB	RP+REB
Zoner	REB	REB	REB	REB	REB	REB	REB	REB	REB	REB

TABLE 6. Resources consumption analysis.

Antivirus	CPU(%)	RAM(%)	Battery (mAH)	Storage Size (MB)
Dr Web	16	56	0.91	7.13
AntiVirus Free	22	65	0.84	4.7
Lookout	18	69	0.9	9.05
Norton	30	60	1	17.15
ESET	21	64	0.98	7.93
Zoner	19	56	0.8	1.56

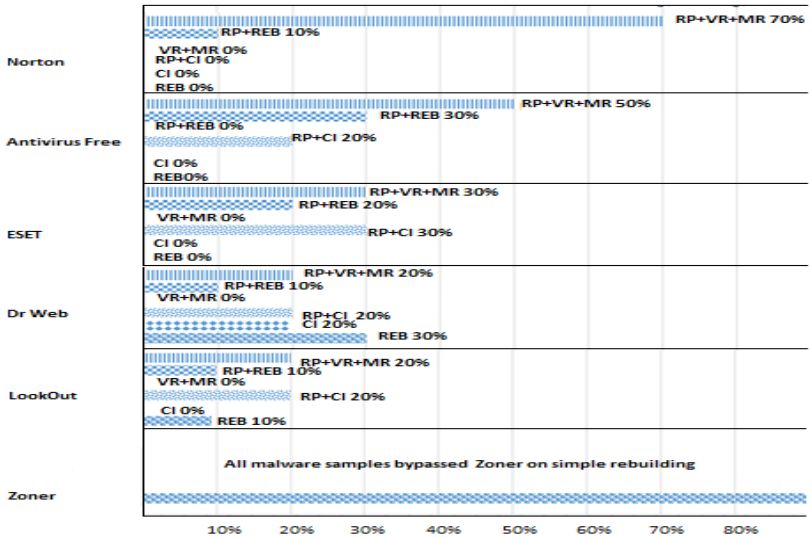

Fig. 3. Experimentation results.

REFERENCES

[1] G. McGraw and G. Morrisett, "Attacking malicious Code: report to the InfoSec research council," *IEEE Software Magzine*, vol. 17, no. 5, Sep.-Oct., 2000.

[2] S. Aurangzeb, M. Aleem, M. A. Iqbal, and M. A. Islam, "Ransomware: A Survey and Trends," *Journal of Information Assurance & Security*, vol. 6, no. 2, 2017.

[3] M. P. Dalla and F. Maggi, "Testing android malware detectors against code obfuscationobfuscation: a systematization of knowledge and unified methodology," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 3, 2017.

[4] "How antivirus works.," [Online]. Available: https://goo.gl/4HxMu1. [Accessed 23 7 2017].

[5] "Android Developers:," [Online]. Available: https://goo.gl/q9sLWI.. [Accessed 22 5 2017].

[6] "Dex2jar," [Online]. Available: http://code.google.com/p/dex2jar/.. [Accessed 22 5 2017].

[7] "Apktool.," [Online]. Available: http://code.google.com/p/apktool/.. [Accessed 22 5 2017].

[8] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," *Department of Computer Science, The University of Auckland*, New Zealand, 1997.

[9] M. Zheng, P. P. Lee, and J. C. Lui, "ADAM: an automatic and extensible platform to stress test android," *in International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer Berlin/Heidelberg.

[10] "VirusTotal," [Online]. Available: https://goo.gl/DlTruF. [Accessed 22 5 2017].

[11] J. Nagra, C. Thomborson, and C. Collberg, "A functional taxonomy for software watermarking," *Australian Computer Science Communications*, vol. 24, no. 1, pp. 177-186, 2002.

[12] . M. Christodorescu and S. Jha, "Testing malware detectors.," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 34-44, 2004.

[13] M. Christodorescu, S. Jha, S. Seshia, . D. Song, and R. E. Bryant, "Semantics-aware malware detection," *IEEE symposium on Security and Privacy*, 2005.

[14] M. Protsenko and . T. Muller, "Pandora applies non-deterministic obfuscation randomly to android.," in *"The Americas"* (MALWARE), 2013.

[15] "Contagio minidump," [Online]. Available: https://tinyurl.com/6b6v7jp. [Accessed 27 5 2017].

[16] A. Zaman and Z. Imtiaz, "MobiBench," *Capital University of Science and Technology*, Islamabad., 2016.