Is Your Smartphone Really Safe? A Wake-up Call on Android Antivirus Software Effectiveness

Andrea Piccione[1,3], Giorgio Bernardinetti[2,3], Alessandro Pellegrini[2,3,*] and Giuseppe Bianchi[2,3]

[1]*Sapienza, University of Rome, Via Ariosto 25, 00185, Roma, Italy*
[2]*University of Rome Tor Vergata, Viale del Politecnico 1, 00133, Roma, Italy*
[3]*National Inter-University Consortium for Telecommunications (CNIT), Viale G.P. Usberti, 181/A, 43124, Parma, Italy*

Abstract

A decade ago, researchers raised severe concerns about Android smartphones' security by extensively assessing and recognising the limitations of Android antivirus software. Considering the significant increase in the economic role of smartphones in recent years, we would expect that security measures are significantly improved by now. To test this assumption, we conducted a relatively extensive study to evaluate the effectiveness of off-the-shelf antivirus software in detecting malicious applications injected into legitimate Android applications. We specifically repackaged seven widely used Android applications with 100 obfuscated malware instances. We submitted the 700 samples to the VirusTotal web portal, testing the effectiveness of the over 70 free and commercial antiviruses available in detecting them. For the obfuscation part, we intentionally employed publicly available tools that could be used by "just" a tech-savvy adversary. We used a combination of well-known and novel (but still simple) obfuscation techniques. Surprisingly (or perhaps unsurprisingly?), our findings indicate that almost 76% of the samples went utterly undetected. Even when our samples were detected, this occurred for a handful (never more than 4) of Android antivirus software available on VirusTotal. This lack of awareness of the effectiveness of Android antivirus is critical because the false sense of security given by antivirus software could prompt users to install applications from untrusted sources, allowing attackers to install a persistent threat within another application easily.

Keywords

Android, Smartphone, Malware, Antivirus, Detection

1. Introduction

Android security is a critical issue today, where mobile devices have become an indispensable part of our daily lives. They are no longer just a communication tool but also a gateway to various online services and applications. From 2020 to 2021, Android phones captured more than 82% of the market share [1].

However, security concerns have also grown with the increased use of smartphones. As the

ITASEC 2023: The Italian Conference on CyberSecurity, May 03–05, 2023, Bari, Italy
*Corresponding author.
piccione@diag.uniroma1.it (A. Piccione); giorgio.bernardinetti@uniroma2.it (G. Bernardinetti); a.pellegrini@ing.uniroma2.it (A. Pellegrini); giuseppe.bianchi@uniroma2.it (G. Bianchi)
0000-0003-1367-2861 (A. Piccione); 0000-0001-6222-0365 (G. Bernardinetti); 0000-0002-0179-9868 (A. Pellegrini); 0000-0001-7277-7423 (G. Bianchi)

number of Android devices grows, so does the risk of malware infections. In 2020, over twenty-five million Android devices got infected by malware [2]. These infections can lead to data theft, loss of privacy, and other serious consequences. One of the primary methods for defending against malware is using antivirus software [3], designed to detect and remove malicious code. However, the increasing sophistication of malware presents a significant challenge for antivirus software to keep pace [4].

One of the fundamental ways malware authors can hide their code from antivirus software is by using obfuscation techniques [5]. These techniques can involve encoding or encrypting the malicious code or hiding it within seemingly benign code. This makes it more difficult for antivirus software to detect and remove the malware [6], allowing it to persist on the device and carry out its malicious activities.

Furthermore, users must be vigilant about downloading only trusted applications from trusted sources. Indeed, the security threat posed by third-party repositories for Android applications is quite significant [7]. Third-party repositories, also known as alternative app stores, offer Android users the ability to download and install applications that may not be available through the official Google Play Store. While these alternative app stores can offer a broader range of applications, they also present many security risks. Indeed, if applications are installed outside Google Play, users automatically waive the use of Google Play Protect [8, 9], which automatically scans all applications installed on a device and compares them with a constantly updated database of known threats.

Ten years ago, researchers had already highlighted the inefficiency of antivirus software for smartphones [10], which raises questions about the overall security of these devices. In the years since then, the economic role of smartphones has significantly increased, and we would expect security measures to have kept pace with this trend.

This research paper aims to re-assess the capabilities of off-the-shelf antivirus software on Android against simple attacks and attack vectors. In particular, we focus on a combination of application repackaging and malware obfuscation to deliver malware on Android devices. We then test our camouflaged malware automatically using Virus Total to determine how many antiviruses can detect the presence of malware.

We used a combination of well-known open-source frameworks [11] and custom steganography-based approaches [12] to conduct our security tests. The results indicate that despite technological advancements, the security of smartphones remains a significant concern, as end users can still be vulnerable to various types of malware.

2. Background

This Section introduces the fundamental architectural building blocks we have leveraged in our experimental assessment.

2.1. The Architecture of an Android App

Java and Kotlin are the main languages used to develop Android applications. These applications are eventually packaged as APKs, a compressed file format containing all the files required for

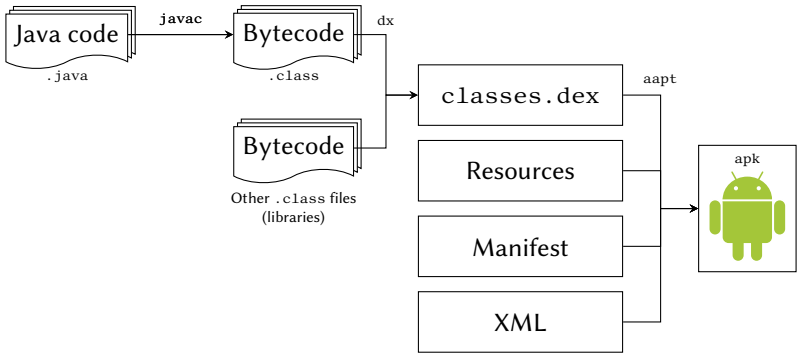

Figure 1: APK Compilation Process

app operation. The organisation of an APK and the compilation process are depicted in Figure 1. The following file formats are included in the app:

- Dalvik Executable (DEX) file: The executable file created by compiling Java source code.
- A manifest file is a file that contains app attributes like rights, the app package file, and the version.
- eXtensible Markup Language (XML) file: A file that defines the user interface (UI) layout and values.
- A resource file is a file that contains resources necessary for program operation, such as photos.

The manifest file is in XML format; other XML files required for program execution are binary encoded. Then, the DEX, XML, manifest, and resource files are ZIP-packed into an APK file. At first, the APK file does not have the developer's signature, which is required for distribution. Using Jarsigner [13], the unsigned APK file may be self-signed with the developer's private key. The developer's signature and public key are then appended to the APK file, completing the Android app development process.

The interesting part about this Android application organisation is that the generation process can be easily inverted. Indeed, the content of the APK can be retrieved directly, being a simple archive.

The format of the DEX file, reported in Figure 2, consists of a series of sections, each containing different types of data. The first section is the header section, which includes information on the file, such as its size, version, and other metadata. The following section is the string section, which contains all the strings used in the app, including class names, method names, and variable names. The string section is followed by the type section, which contains information about the types used in the app, such as class names and method signatures.

The bulk of the DEX file comprises the code section, which contains the compiled bytecode for all the methods in the app. Each method is stored in its section, along with information about its exception handlers and other metadata. Finally, the file ends with several optional sections, including the debug information section, the annotation section, and the class definition section, which contains information about the classes used in the app.

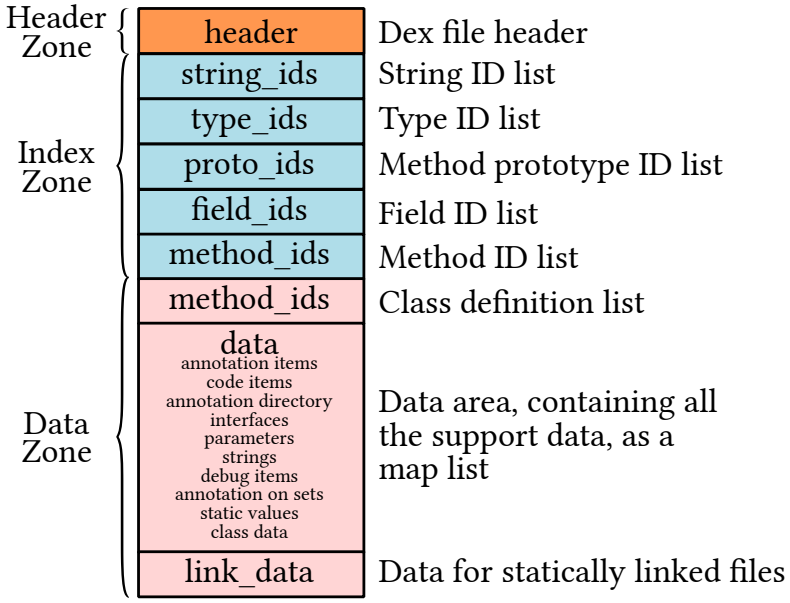

Figure 2: DEX File Organisation.

2.2. Android App Components: Activities and Broadcast Receivers

We leverage two fundamental elements of Android applications in our assessment: *activities* and *broadcast receivers*.

An Android activity is a component representing an Android app's single screen or user interface. Each activity carries out a specific action or duty, such as presenting a list of things, collecting user input, or showing a map. An Activity allows users to engage with the app and display the app's content on the screen. An Android app can have numerous activities, with the system managing the flow between them based on user involvement. When a user launches an app, the primary activity is launched, and other activities may be started and terminated as the user interacts with the program.

Conversely, an Android broadcast receiver is an operating system component that enables applications to respond to system-wide broadcasts. It is a listener for broadcast events such as app installation or removal, SMS message reception, or changes in the device's connection state. When a broadcast event happens, the Android system sends an Intent message to any registered Broadcast Receiver, which can then reply. A Broadcast Receiver's goal is to allow applications to receive alerts and respond to events without needing to be active in the foreground.

2.3. Tampering with Existing Android Applications

Looking at the organisation of the DEX file format, it is clear that exporting the Java/Kotlin classes is trivial. From there on, reconstructing the original source code is a matter of decompiling the Java bytecode, which can be accomplished by relying on standard tools [14]. The interesting point is that they can be arbitrarily altered once the source files are obtained. Then the whole

APK generation process can retake place, thus generating a newly-packaged application.

This process can be automated using automated tools (e.g., [15]). In the repackaging loop, any code and resource manipulation can take place [11]. For example, the original symbols in the source can be renamed, the control-flow graph can be reordered, assets can be encrypted, or metadata can be refactored. All in all, these techniques enable changing the hash of the files embedded in the APK (to confuse static scanners) or fool n-gram analysis [16]. While packers can have legitimate uses [17], they are a significant security threat to Android devices.

Indeed, a severe risk posed by APK injection is that the modified app may continue to function seemingly normally, making it difficult for the victim to detect that their device has been compromised. This can allow the attacker to persist on the device for an extended period, increasing the scope and impact of their malicious activities.

3. Related Work

The ease of perpetrating attacks by relying on malware repackaged into applications has already been studied in the literature. In [18], the authors have examined some of the top Android-based smartphone banking applications in Korea available on the Android Market or a third-party market to see if a money transfer might be completed to an unexpected recipient. This work demonstrated that such an attack is achievable without unlawfully obtaining any of the sender's personal information, such as the sender's public key certificate, the password to their bank account, or their security card. Differently from this work, the authors focus on a specific kind of app and concentrate more on the source of the vulnerability. Conversely, we want to show the ease of bypassing antivirus security through a well-known technique such as repackaging.

A study of Android's security implications and pitfalls has been presented in [19]. In this work, the authors have classified many potential threats and attack vectors that affect the Android system. Similarly, a comprehensive analysis of the (re)packaging techniques has been presented in [20]. In this work, the authors propose and exercise a dynamic analysis system based on the Android system and Linux kernel hooking that allows inspecting the behaviour of packers. Our work is orthogonal, as we study the effect of such evasion techniques and potential threats on off-the-shelf antivirus systems.

In [21], the authors perform an empirical investigation of over 15,000 applications to get insights into the elements that promote the growth of repackaged applications and concentrate on the reasons that could tempt end users into installing repackaged applications. We consider our work orthogonal because we study how much a repackaged app infected by malware can bypass antivirus security.

A work sharing similar goals can be found in [10]. Here, the authors study how different obfuscation techniques affect the detection rate of antivirus software. In this work, we conduct a similar experiment, which allows observing to what extent the security panorama has improved since that seminal work was published ten years ago. We also consider an additional obfuscation technique that was not available at that time.

4. Experimental Evaluation

4.1. Testbed setup

In our experimental setup, we have used a combination of regular applications and malicious payloads. Regarding the applications to repackage, we have selected 7 that can be of daily use for different classes of common users, with a non-minimal number of installations. They include:

- *CPU-Z*, a popular free program that reports on device information, with more than 50 million installations from Play Store.
- *Cx File Explorer*, a file management app to manage files on your mobile device, PC and cloud storage; it offers storage usage analysis and is installed by more than 10 million users from Play Store.
- *F-Droid*, a catalogue of free applications for Android, a very relevant application since it opens the door to installations from third-party repositories and is unavailable on the Play Store.
- *MyTV+* is a video streaming app that provides access to many channels.
- *ZArchiver*, an archive management app, which is relevant because it provides functionality typically unavailable natively on Android systems, with over 100 million installations from the Play Store.
- *DiskDigger photo recovery*, an app to restore and recover lost photos, images or videos from the internal memory or external memory card, with over 100 million installations from the Play Store.
- *Open Camera*, a completely free camera app with more than 50 million installations from the Play Store.

Concerning the malicious payloads, we have used 100 malware freely available from VxUnderground. They are representative of different attacks that could be perpetrated on mobile devices. We have carefully selected immediately-working malware, as our analysis revealed that a significant percentage (∼4%) of payloads do not expose a main activity. While launching a service rather than an activity requires a similar logic, we chose to avoid malware that lacked a main activity for this evaluation. We tested all malware against VirusTotal, and the detection rate without any manipulation was 100%.

4.1.1. Building Repackaged Malware

We leverage the approach depicted in Figure 3 to embed malware in non-malicious applications. Recalling the organisation of an APK file represented in Figure 1, the first steps consist in decoding the archive to retrieve its components—mainly the decompiled bytecode and the Android Manifest—in a human-readable form. This is done by relying on apktool [22]. This step also allows us to retrieve the name of the payload activities and services used in the final repackaged application to satisfy the constraints imposed by the Android Manifest file.

The malware code is then obfuscated, relying on the well-known ObfuscAPK [11]. This first obfuscation phase aims to hide easily identifiable features of the malware, such as specific string

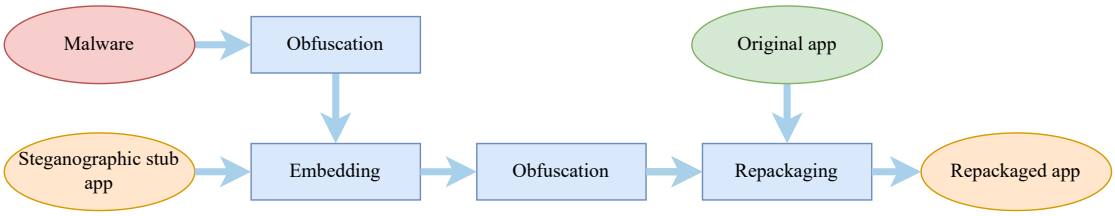

Figure 3: Our Repackaging Process at a Glance.

or image resources. To improve the obfuscation, we extract the original `classes.dex` file from the APK. This is encoded via steganography in a selection of PNG icon assets of a custom-built stub Android application. We use two bits per pixel, selected in a randomised pattern. The encoding process ensures that the original data is preserved, allowing the application to remain functional.We then obfuscate again our stub application to render the logic that unpacks the malware at runtime less identifiable.

The doubly-obfuscated malware is then included in the APK of the original application. This is done by relying on an approach similar to the one proposed in [15]. In particular, the original application is similarly decompiled. The malware stub is attached to an *activity* or a *broadcast receiver*, selected from those available in the original application.

By using activities and broadcast receivers, we are mimicking the will of an attacker to execute a malicious payload exactly when the user is performing some specific event. For the purpose of our experiment, we either use the main activity or the power supply connection broadcast. In this way, we can exercise the capability of the antivirus to detect the presence of obfuscated malware at startup or after some time from the application start.

Our stub application has a simple main activity that retrieves the assets and, using a `DexClassLoader`, instantiates instances of the payload classes at runtime. The payload activity can then be triggered by firing an Intent.

4.2. Results

We have repackaged the applications introducing the obfuscated malware and submitted them to the VirusTotal web portal, which is connected to various free and commercial antiviruses at its backend. In this way, we have been able to determine the current detection capabilities of off-the-shelf antiviruses against simple obfuscation activities. We note that we have not differentiated between obfuscated and repackaged malware. Indeed, we have directly applied the process depicted in Figure 3. This is because the purpose of our work is to experiment with known techniques to assess the capabilities of antivirus software, rather than investigating the root cause of possible malware detection.

In Figure 4, we report the percentage of repackaged malware generated in our assessment that has been detected by the antivirus software hosted on Virus Total—each sample evaluated was a combination of malware injected into benignware APK. As can be seen, 76% of the samples are not detected by any antivirus, while 15% by only one. 2 or more antivirus applications detect the remaining part, but surprisingly there is no sample with 5 or more detections. In

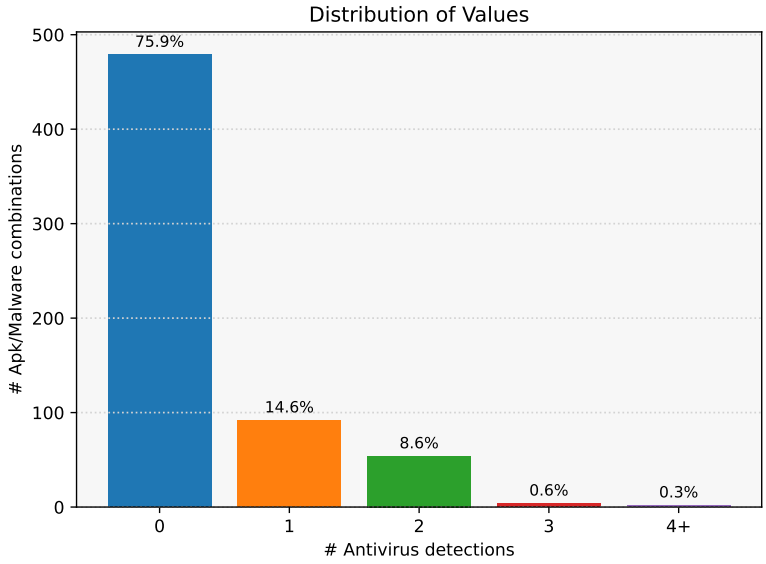

Figure 4: Detection Rate for the Generated Obfuscated Malware.

Table 1
Most Detected Malware Samples

Malware	Detections
HEUR-Trojan-SMS.AndroidOS.Opfake.bo v1	6
HEUR-Trojan-SMS.AndroidOS.Opfake.bo v2	6
HEUR-Trojan.AndroidOS.Boogr.gsh	5
HEUR-Trojan-Spy.AndroidOS.SpyNote.ar	5
HEUR-Backdoor.AndroidOS.Climap.a	4
HEUR-Trojan.AndroidOS.Piom.akwz	4
HEUR-Trojan.AndroidOS.Timethief.b	4
HEUR-Trojan.AndroidOS.Agent.lp	4
HEUR-Trojan-Spy.AndroidOS.Campys.a	4
HEUR-Trojan-Banker.AndroidOS.Ermak.a	4

total, the number of antiviruses used is more than 70. This means that, in general, more than 90% of the available antiviruses may not be effective in detecting known threats if they are manipulated with a careful sequence of known obfuscation techniques. This leaves Android users vulnerable to security breaches. In order to understand whether there is a bias towards specific malware being detected, i.e. whether certain payloads are more easily detected and thus cause an increase in detection, in Table 1 we list the malware that, in any combination with a legitimate APK, has been detected (i.e., independently of the benignware APK into which it was injected). From the results, we observe that the number of threats that are detected many times is minimal. Therefore, it can be concluded that the distribution of detections by antiviruses reported in Figure 4 is not affected by particularly more easily detectable payloads.

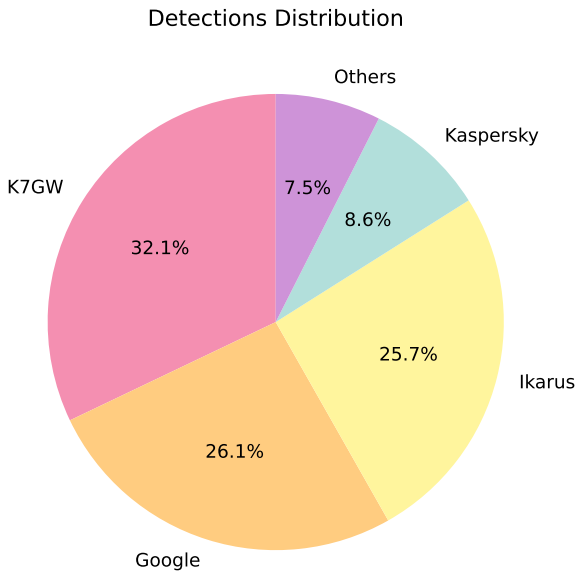

Figure 5: Detection Rate by the antiviruses in VirusTotal pool

Regarding the selected applications, the most detected APK is *DiskDigger photo recovery*—it has been detected 86 times in our experiment. By looking at the internals of this application, we noticed that it is internally using some heavy obfuscation techniques, probably in an attempt to protect the implementation of the application. Therefore, this autonomous obfuscation is why antivirus heuristics suspect the application is malicious. Indeed, we suspect that (since it is highly downloaded) antivirus software has included an exception signature for the original APK. This consideration means that, in reality, the results in Figure 4 are higher than they would be in case an attacker would cherry-pick a proper APK to inject its malware in.

Finally, we report in Figure 5 the share of the detection rate by the different antiviruses in the VirusTotal pool. As can be seen, the best-performing ones are the K7 antivirus, which focuses on more robust generics and heuristics, and Google, which implements techniques shared with the aforementioned Play Protect service. This is an additional indication of the effectiveness of relying on third-party repositories since the highest detection rate comes from techniques implemented in Play Protect.

5. Conclusions

Ten years after the alarming results documented in 2013 regarding the inefficacy of Android antivirus [10], we would have expected today a significant improvement in the security of Android smartphones. However, our research has shown that the security of Android smartphones is still at risk, especially for ordinary users who rely on cheap commercial antivirus software.

More specifically, we conducted a large-scale experiment by submitting 630 applications

repackaged with malware, created by combining seven widely used applications with 100 obfuscated malware instances, to the VirusTotal web portal. We evaluated the effectiveness of the various free and commercial antiviruses (more than 70) available on its backend. Our results showed that almost 76% of the samples were not detected by any single antivirus, and more than 90% of the antivirus did not detect any repackaged sample. What specifically worried us is that we did not use, in our experiments, “too" advanced obfuscation techniques. Still, we used just a careful combination of well-known techniques with basic steganography approaches.

In conclusion, our study explicitly highlights the significant security threat posed by repackaging Android applications to distribute harmful applications that evade antivirus software detection. The ease with which this can be done raises questions about the effectiveness of the level of protection provided to smartphone users, particularly those who may not have the technical expertise and may lack awareness about the actual efficacy of Android antivirus. For such users, simply having an antivirus installed is enough to create a false sense of security and lead them to eventually download applications from untrusted sources, enabling attackers to inject a persistent threat into their devices easily.

References

[1] L. Goasduff, Gartner says global smartphone demand was weak in third quarter of 2019, https://www.gartner.com/en/newsroom/press-releases/2019-11-26-gartner-says-global-smartphone-demand-was-weak-in-thi, 2019. Accessed: 2023-2-6.

[2] P. Bhat, K. Dutta, A survey on various threats and current state of security in android platform, ACM Comput. Surv. 52 (2019) 1–35. doi:10.1145/3301285.

[3] M. Elingiusti, L. Aniello, L. Querzoni, R. Baldoni, PDF-Malware detection: A survey and taxonomy of current techniques, in: A. Dehghantanha, M. Conti, T. Dargahi (Eds.), Cyber Threat Intelligence, Springer International Publishing, Cham, 2018, pp. 169–191. doi:10.1007/978-3-319-73951-9_9.

[4] S. Acharya, U. Rawat, R. Bhatnagar, A comprehensive review of android security: Threats, vulnerabilities, malware detection, and analysis, Security and Communication Networks 2022 (2022). doi:10.1155/2022/7775917.

[5] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, K. Zhang, Understanding android obfuscation techniques: A large-scale investigation in the wild, in: R. Beyah, B. Chang, Y. Li, S. Zhu (Eds.), Security and Privacy in Communication Networks, volume 254 of *LNICST*, Springer Verlag, Cham, Switzerland, 2018, pp. 172–192. doi:10.1007/978-3-030-01701-9_10.

[6] S. Alam, I. Sogukpinar, DroidClone: Attack of the android malware clones - a step towards stopping them?, Computer Science and Information Systems 18 (2020) 67–91. doi:10.2298/CSIS200330035A.

[7] Y. Y. Ng, H. Zhou, Z. Ji, H. Luo, Y. Dong, Which android app store can be trusted in china?, in: 2014 IEEE 38th Annual Computer Software and Applications Conference, ieeexplore.ieee.org, 2014, pp. 509–518. doi:10.1109/COMPSAC.2014.95.

[8] E. Cunningham, Keeping you safe with google play protect, https://cloud.google.com/

blog/products/android-enterprise/keeping-you-safe-google-play-protect/, 2017. Accessed: 2023-2-6.

[9] T. Cho, H. Kim, J. H. Yi, Security assessment of code obfuscation based on dynamic monitoring in android things, IEEE Access 5 (2017) 6361–6371. doi:10.1109/ACCESS.2017.2693388.

[10] M. Zheng, P. P. C. Lee, J. C. S. Lui, ADAM: An automatic and extensible platform to stress test android anti-virus systems, in: Detection of Intrusions and Malware, and Vulnerability Assessment, Lecture notes in computer science, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 82–101. doi:10.1007/978-3-642-37300-8_5.

[11] S. Aonzo, G. C. Georgiu, L. Verderame, A. Merlo, Obfuscapk: An open-source black-box obfuscation tool for android apps, SoftwareX 11 (2020). doi:10.1016/j.softx.2020.100403.

[12] S. Badhani, S. K. Muttoo, Evading android anti-malware by hiding malicious application inside images, International Journal of System Assurance Engineering and Management 9 (2018) 482–493. doi:10.1007/s13198-017-0692-7.

[13] Oracle Corporation, jarsigner, 2018.

[14] J. Hamilton, S. Danicic, An evaluation of current java bytecode decompilers, in: Proceedings of the 9th International Working Conference on Source Code Analysis and Manipulation, SCAM '09, IEEE, Piscataway, NJ, USA, 2009, pp. 129–136. doi:10.1109/SCAM.2009.24.

[15] A. Salem, F. F. Paulus, A. Pretschner, Repackman: a tool for automatic repackaging of android apps, in: Proceedings of the 1st International Workshop on Advances in Mobile App Analysis, A-Mobile 2018, Association for Computing Machinery, New York, NY, USA, 2018, pp. 25–28. doi:10.1145/3243218.3243224.

[16] A. Shabtai, Y. Fledel, Y. Elovici, Automated static code analysis for classifying android applications using machine learning, in: Proceedings of the 2010 International Conference on Computational Intelligence and Security, CIS '10, IEEE, Piscataway, NJ, USA, 2010, pp. 329–333. doi:10.1109/CIS.2010.77.

[17] S. Drape, Intellectual property protection using obfuscation, Technical Report CS-RR-10-02, Oxford University Computing Laboratory, 2011.

[18] J.-H. Jung, J. Y. Kim, H.-C. Lee, J. H. Yi, Repackaging attack on android banking applications and its countermeasures, Wireless Personal Communications 73 (2013) 1421–1437. doi:10.1007/s11277-013-1258-x.

[19] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, C. Glezer, Google android: A comprehensive security assessment, IEEE security & privacy 8 (2010) 35–44. doi:10.1109/MSP.2010.2.

[20] Z. Dong, H. Liu, L. Wang, X. Luo, Y. Guo, G. Xu, X. Xiao, H. Wang, What did you pack in my app? a systematic analysis of commercial android packers, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Association for Computing Machinery, New York, NY, USA, 2022, pp. 1430–1440. doi:10.1145/3540250.3558969.

[21] K. Khanmohammadi, N. Ebrahimi, A. Hamou-Lhadj, R. Khoury, Empirical study of android repackaged applications, Empirical Software Engineering 24 (2019) 3587–3629. doi:10.1007/s10664-019-09760-3.

[22] R. Winsniewski, apktool: A tool for reverse engineering android apk files, 2012.