On the Feasibility of Android Stegomalware: A Detection Study

Diego Soi[1,*], Silvia Lucia Sanna[1], Angelica Liguori[2], Marco Zuppelli[2], Leonardo Regano[1], Davide Maiorca[1], Luca Caviglione[2], Giuseppe Manco[2] and Giorgio Giacinto[1]

[1]*University of Cagliari, Italy*
[2]*National Research Council, Italy*

Abstract
Android malware represents an evolving threat within the modern cybersecurity landscape due to the increasing importance of mobile systems in everyday life. Obfuscation and source code manipulations are systematically employed to bypass security measures and improve the effectiveness of attacks, especially to prevent detection or endanger the privacy of users. However, they represent only a portion of the evasive techniques that can be employed to make malicious software stealthier. In this work, we showcase a prime assessment of the joint use of steganography and repackaging techniques to hide information within Android APK resources. Specifically, we assess the capabilities of real-world antivirus aggregated by VirusTotal to identify payloads cloaked within audio and images of 20 popular Android applications. Our investigation demonstrated that repackaging steganographically modified assets is not always possible. Besides, our results revealed that common antivirus are not able to identify applications containing hidden data, thus highlighting the need for new Indicators of Compromise.

Keywords
Android Stegomalware, Steganography, Evasion

1. Introduction

The Android Operating System (OS) is the main OS employed for mobile devices (e.g., smartphones, tablets, and smartwatches), and it supports a wide range of functionalities. Modern devices extend the basic operations (i.e., messaging and phone calls), to include security-sensitive tasks, such as multi-factor authentication or home banking. Hence, malicious software that can perform various harmful activities poses a significant risk to the security and privacy of users. In general, threats are mitigated by employing antimalware detection systems, which typically rely on static analysis methods, preventing the need to use an emulator or a real device [1]. Unfortunately, an attacker can leverage the static analysis limitations of antimalware systems by concealing malware payloads. In this way, it is possible to deceive the analysis and evade antivirus (AV) detection and security checks. The literature abounds with works addressing the challenge arising from attackers deploying software manipulation through code injection [2, 3] or obfuscation [4, 5]. However, an aspect often neglected concerns the impact of new and advanced cloaking techniques to make Android malware stealthier. Therefore, this work wants to shed new light on attackers empowering next-generation Android malware with information-hiding capabilities, especially through the use of digital media steganography. At the same time, it wants to update the investigation performed in [6], which is now a decade old and focused on earlier forms of Android malware. In more detail, this work outlines the following research questions:

- **RQ1**: owing to the increasing complexity of modern mobile applications as well as the presence of

Joint National Conference on Cybersecurity (ITASEC & SERICS 2025), February 03-8, 2025, Bologna, IT
*Corresponding author.
diego.soi@unica.it (D. Soi); silvial.sanna@unica.it (S. L. Sanna); angelica.liguori@cnr.it (A. Liguori); marco.zuppelli@cnr.it (M. Zuppelli); leonardo.regano@unica.it (L. Regano); davide.maiorca@unica.it (D. Maiorca); luca.caviglione@cnr.it (L. Caviglione); giuseppe-manco@cnr.it (G. Manco); giorgio.giacinto@unica.it (G. Giacinto)
0009-0009-0092-9067 (D. Soi); 0009-0002-8269-9777 (S. L. Sanna); 0000-0001-9402-7375 (A. Liguori); 0000-0001-6932-3199 (M. Zuppelli); 0000-0002-9259-5157 (L. Regano); 0000-0003-2640-4663 (D. Maiorca); 0000-0001-6466-3354 (L. Caviglione); 0000-0001-9672-3833 (G. Manco); 0000-0002-5759-3017 (G. Giacinto)

runtime checks enforced by developers to prevent unauthorized alterations (e.g., rebranding) [7, 8], is it possible to repack an Android application with malicious steganographic contents?
- **RQ2**: the increasing degree of effectiveness of security tools could partially fail when facing cloaked or advanced threats. Hence, does the steganographic embedding scheme have an impact on the results reported by the most popular antivirus?

To answer such questions, we performed a set of experiments to explore the possibility of repackaging well-known applications containing steganographic malicious content and to test whether current AVs are able to spot their presence. To this aim, we performed experiments by selecting the 20 most downloaded apps on the Google Play Store, and we repacked them to include the payload of `Necro`, a well-known trojan that recently infected 11 million Android devices [9]. Among the various apps, `Necro` has spread mainly via a modded version of the Spotify audio player distributed across unofficial channels. To perform experiments, we embedded part of the trojan payload through two variants of Least Significant Bit (LSB) steganography in the audio and image resources of the target apps. Finally, we tested the repacked apps with VirusTotal (VT) to assess the ability of its AV engines to detect the various malicious steganographic payloads. As specified in Section 2.2, according to the Android documentation *assets* and *resources* are two different objects inside an APK due to the compiling (i.e., APK building) and access mechanism, during application execution. In the following, we will interchange the terms *assets* and *resources* unless doubts arise.

Therefore, the contributions of this work are threefold: *i*) it investigates whether modern Android applications should be considered a valuable target for attackers deploying information-hiding capable schemes; *ii*) it provides a prime static assessment of real-world security frameworks when facing malicious information cloaked within digital media; *iii*) it showcases the main research gaps, focusing on publicly available detection tools (i.e., VirusTotal), that need to be filled to advance the security posture of mobile applications.

The rest of the paper is structured as follows. Section 2 provides background information on steganographic techniques observed in real-world stegomalware and core technical details on Android applications, whereas Section 3 reviews prior research on the mitigation of mobile threats. Section 4 presents the methodology employed to investigate stegomalware targeting the Android ecosystem, and Section 5 showcases quantitative results. Finally, Section 6 concludes the paper and hints at possible future research directions.

2. Background

This section provides details on mechanisms for endowing Android malware with hiding capabilities. First, it introduces the main steganographic methods observed in real-world malicious software. Then, it presents technical details and functional constraints of Android applications.

2.1. Malware Using Steganography

Modern malware is increasingly using steganographic techniques to conceal malicious payloads within seemingly innocuous digital files, such as images, audio files, or documents. This new wave of threats, often referred to as *stegomalware*, deploys steganography to avoid signature-based detection, implement multi-stage load architectures, or covertly extract sensitive information without being blocked by firewalls or network intrusion detection systems [10, 6]. Among all the possible objects that can be used, digital images are the most frequently exploited carriers: a recent review of stegomalware cases covering media, text, and network steganography revealed that images are the preferred choice to embed malicious data [11].

The hiding process should remain as straightforward as possible to prevent the introduction of detectable signatures (e.g., inflated loading times or major execution overheads). Consequently, modern malware often utilizes spatial domain-based image steganography, where the pixels of the images are directly manipulated to embed the information. One of the most common techniques employed by attackers is the LSB steganography. It consists of concealing secret data by overwriting the least significant bits of the color components of the pixels of the carrier image. The adoption of LSB steganography to hide information within digital images has been observed in many real-world malware samples, such as `PNGLoader`, `RDAT`, and `RegDuke` [10, 11]. Moreover, some malicious software introduced alternative cloaking schemes able to exploit multiple bits of the color space: this allows for better tradeoffs in terms of detectability versus the volume of cloaked data. For instance, malware like `PowLoad` and `Ursnif` take advantage of the `Invoke-PSImage`, an LSB technique implemented in PowerShell which embeds the malicious payload within the 4 least significant bits of the green and the blue channels. Other variants developed by `OceanLotus` exploit 2/3 bits according to the targeted color channel [10].

2.2. Android Applications (Apps)

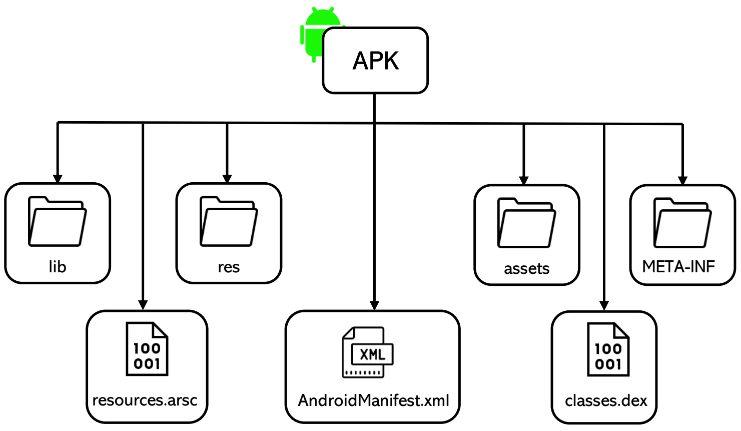

Figure 1: Structure of an application with all required resources.

An Android application is a software written in Java or Kotlin designed to run on Android devices (e.g., smartphones, tablets, and smart TVs). The kernel of the Android OS is based on the Linux core, and native written in C/C++ is imported to interact with the main native activities and hardware components (e.g., camera and microphone).

Android applications are packed into a file called APK (Android Package). As depicted in Figure 1, it consists of different resources required to be installed and executed: *i*) `lib` directory, holding the libraries for each CPU configuration (i.e., ARM and $x86$ emulation, both for 32 and 64 bits); *ii*) `assets` and `res` directories, holding all resources (e.g., images, audio, and XML files) employed by the graphical interface; *iii*) `META-INF` directory, holding manifest information and other signature metadata; *iv*) `AndroidManifest.xml`, containing all permissions (i.e., required rights to use restricted APIs), and app entry points; *v*) `.dex` (Dalvik EXecutable format) files holding the code executed by the ART (Android Runtime) whose framework extracts and loads the needed resources through the identifier (e.g., `R.drawable.image_id`) at runtime; *vi*) `resources.arsc` file containing all pre-compiled resources, such as XML files. All the aforementioned assets are then built by using development tools such as Android Studio to obtain a compiled APK signed with the developer key.

To better comprehend the focus of this work, it is important to understand how resources are stored in the APK and used at runtime as well as how they are handled in memory and integrated with the code. Resources in Android are files and static contents compiled and integrated into the APK, while assets are uncompiled files stored as-is and programmatically accessed. During compilation, resources specific to different device configurations are included in the `res` directory, which contains different self-explainable subdirectory names according to the content, type and use of the resource, such as *animator*, *color*, *font*, *layout*, *xml*, *raw* for the raw files usually audio, *drawable*, *mipmap* for the launcher icon densities. At runtime, the system extracts and loads the resource that matches the current configuration, such as screen size, by using an assigned identifier accessed through the `R` class and XML file.

Android applications can be easily reversed by employing decompiling tools, e.g., `ApkTool`[1] or `Jadx`[2], which allow accessing the original resources and code. The latter is in the form of smali code, a representation of the Android APK Dalvik code, or Java/Kotlin that is the typical language used to develop Android applications. Yet, the original app can still be modified by injecting, removing, or altering resources or code (often embedding malicious payloads) to be then re-built with `ApkTool`. In essence, such a process can be summarized in the following four main steps: *i*) *decompile* the application to extract the Java code and all the resources; *ii*) *modify* the resource files via the different techniques; *iii*) *rebuild* all the files and obtain the new APK; *iv*) *re-sign* the APK with a new certificate, as the original signing key of the legitimate developer is not accessible. For this reason, the main changes in a repacked APK are the signature, code, and resource structure according to what we have added/deleted, as well as the `AndroidManifest.xml` in case of newly added permissions. As it will be detailed later in Section 5.2, the repackaging stage can fail due to the presence of anti-tampering techniques, which account for errors during the compilation of resources or code.

3. Related Work

Over the years, Android malware detection has emerged as a significant challenge due to the critical security vulnerabilities associated with Android smartphones. The employed approaches differ by analysis type and, consequently, by features or IoCs considered. Indeed, three main detection techniques can be outlined: *i*) *static* methods, which parse the files of the applications to extract strings (e.g., URLs and IPs), code patterns (e.g., API calls, smali opcodes), and permissions requested at runtime by the APK [12, 13]; *ii*) *dynamic* methods, which run the applications in a testing environment searching for API calls and network flows [14]; *iii*) *hybrid* methods, combining dynamic and static ones [15]. While dynamic analysis provides deeper insights into malware behavior, it comes with intrinsic limitations, such as extended time requirements for a comprehensive analysis or the risk that malware remains latent when executed in a sandboxed environment [16]. Thus, static analysis remains a core component of most detection systems.

Furthermore, two additional detection strategies have proven to be effective. The first entails *signature-based* methods [17], which rely on identifying known patterns in the extracted features with the limitation of clearly detecting only known malware. The second takes advantage of techniques employing Machine Learning (ML) [18] and Deep Learning (DL) approaches [19], which can identify both known and unknown malware by learning patterns from a training set. Learning algorithms are used with static [20], dynamic [21] or hybrid analysis [22]. Besides, some malware uses adversarial attacks on ML and DL algorithms to evade malware detection [5, 23].

Recalling that repackaging can be used to modify resources/code and rebuild an application, embedding malicious code into legitimate Android applications can decoy ML algorithms [24]. In fact, repackaging significantly affects the detection capabilities of ML algorithms, as these techniques can obscure signatures or static features that many detection systems rely on. For instance, subtle modifications of the structure of an app can lead ML-based models to misclassify malware as safe, demonstrating the vulnerability of certain ML algorithms to adversarial manipulation. A new technique based on analyzing the dynamic user interfaces has been developed to detect repackaged Android apps [25], while new strategies to protect applications from being repacked have been studied [7]. To this extent, the app's integrity is checked at runtime by encrypting bytecode sections, subsequently decrypted with a key derived during execution.

As shown, the literature dealing with Android malware detection largely neglects threats using steganography. A notable exception is [6], where authors preliminary investigated whether steganography can be used to distribute malicious content via the Google Play store. Another work focusing

[1]ApkTool: https://apktool.org/
[2]Jadx: https://github.com/skylot/jadx

on information-hiding instead of "classical" obfuscation techniques is [26], which proposes to use high-level indicators to spot the presence of malicious applications bypassing sandboxes via covert communications. Another recent work focused on Android stegomalware by investigating techniques to hide malicious dex files in the images of an application [27]. Yet, many works addressing the mitigation of stegomalware in desktop scenarios are starting to emerge. For instance, the work in [28] tackles the problem of revealing the presence of data hidden through LSB techniques within the in-line objects of HTML content. An analysis of the popular `Invoke-PSImage` mechanism revealed that it introduces a major statistical signature in targeted pixels, which can be exploited to prepare trivial detection frameworks [29]. In some malware (e.g., `MiniDuke` and `Stegoloader`), data is cloaked without using steganography but by prepending/appending the malicious contents to an image. In this case, the data can be revealed (or removed) by searching for inconsistencies in the size of the file or the preamble/trail sequence of the image format [30].

With the advent of artificial intelligence, more advanced detection solutions have been also proposed. For instance, [31] introduces a framework that employs machine learning methods, including Linear Discriminant Analysis, Random Forest, and Back-Propagation Networks, specifically for detecting malicious `PowerShell` scripts. In [32], the authors propose an AI-based framework that not only detects stegomalware attacks but also classifies their specific type.

Lastly, detection can be complemented by sanitization, a countermeasure technique aimed at destroying the embedded malicious code without compromising the original quality of the carrier. Traditional sanitization approaches, such as additive noise, lossy compression, and image filtering [33, 34], are effective at disrupting hidden content but may compromise the quality of the carrier. To overcome this issue, sophisticated approaches leverage artificial intelligence. For example, in [35, 36], the authors propose neural architectures for removing the embedded malware payloads while maintaining the integrity of the carrier, i.e., the image.

4. Methodology

This section deals with the methodology used to assess the impact of stegomalware on the Android ecosystem. First, it introduces the dataset of steganography resources. Second, it showcases how the various malicious contents have been repackaged to produce malicious applications. Lastly, it presents the experimental testbed.

4.1. Dataset Preparation

To create the dataset, we first computed some statistics about the resources contained in real-world Android applications downloaded from the Google Play Store. Specifically, we wanted to understand the most common formats for images (i.e., pixel dimension, extension, channel type) and audio (i.e., duration, number of channels, and extension) in order to build a dataset of steganographically-modified assets to be included within repacked applications.

To hide secret information within audio files, we used the `AudioStego` tool[3], which allows to hide information within both MP3 and WAV audio files. In essence, `AudioStego` converts the text to be hidden into its binary ASCII representation. Then, it embeds each bit of the secret message into the LSB of the audio samples that compose the file. Since the amount of audio samples that can be used to conceal the payload is limited, we hide a short list of 3 malicious URLs obtained from the URLhaus repository[4]. The URLs have been separated through the `*d*` escape sequence. This "template" has been observed in various malware campaigns (see, e.g., [11], and the references therein) and allows an attacker to search for URLs by directly looking at the escape characters sequence, thus making the "extraction" process simpler.

[3]AudioStego: https://github.com/danielcardeenas/AudioStego/tree/master
[4]URLhaus: https://urlhaus.abuse.ch/

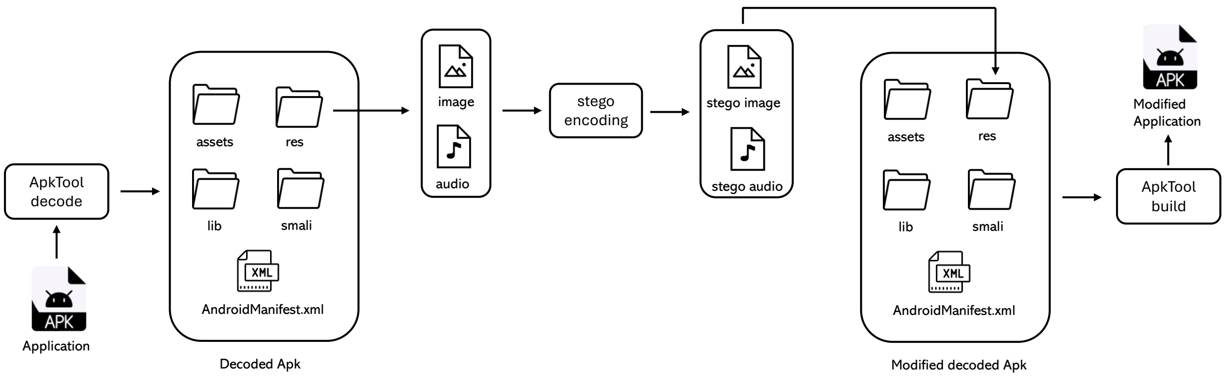

Figure 2: Workflow of the methodology employed to inject strings into APK raw resources and rebuild the application.

Concerning steganographic techniques used for images, we considered two LSB-based strategies [10]. Specifically, *i*) LSB plain in which the payload is hidden in the first least significant bit of each color channel, i.e., red, green, and blue; and *ii*) LSB variant where the payload is hidden in the three least significant bits of the red and green channels and in the two least significant bits of the blue channel. This technique has been extensively observed in the `OceanLotus` advanced persistent threat [11]. In general, images allow for hiding larger contents with respect to audio files. Without loss of generality, in this work, we hide only a portion of the `Necro` trojan payload. In fact, many modern malware distribute their functionalities leveraging multi-stage loading paradigms, reducing the likelihood of being detected by defensive mechanisms [37] and the possible "visual" artifacts generated in the resulting images. Moreover, to analyze an even more realistic and complex scenario, we used two different hiding patterns: *sequential* and *square*. In the sequential strategy, the payload is hidden by starting from the first pixel of the image, while in the square pattern, the payload is concealed within equally-sized blocks arranged diagonally across the image. We point out that it is not always possible to apply both strategies/patterns to all the images composing our dataset. For example, 8-bit greyscale images have a single color channel, making it impossible to apply the LSB variant method.

4.2. Repackaging Workflow

The methodology employed to create the dataset of stegomalware samples is outlined in Figure 2. Specifically, each application is decoded using `ApkTool` to obtain the original resources, as explained in Section 2.2. Then, the steganographic assets were placed in the same locations with the original names, allowing the applications to be correctly repacked.

To avoid burdening the encoding/repack workflow, we limited our injection to a maximum of two images and two audio files per application, whenever feasible. These assets were selected to meet the specific requirements of the steganography methods used, such as the number of channels, image formats (e.g., RGB and greyscale), and audio characteristics like file type (e.g., MP3) and duration (around 2 seconds). By applying the techniques outlined in Section 4.1, we created each asset with multiple information-hiding techniques, enabling us to generate a wide range of combinations for comprehensive coverage. However, to create the various apps, we did not consider all the combinations of malicious resources for two main reasons. First, the resources maintain the name, so it would be overwritten if we consider the same resources modified with different techniques. Second, even if we renamed the resource to avoid overwriting, the resource would not have been employed in any file describing the application layout. Indeed, as described in Section 2.2, all the resources must be declared and called inside the source code to guarantee a fair assessment of security implications.

After repackaging, each app was signed with different signed keys not to correlate the signing key with the detection result. This is because VT can associate the detection result to the signing key instead of the real app analysis. Moreover, the different signing keys have been used to understand if the applications could be correctly run within an emulator for testing purposes, as done in the real world where each app is signed by different developers with their own signing key. The source code of our testing pipeline can be found in the GitHub repository[5].

4.3. Experimental Testbed

To quantify how security tools are effective in detecting the steganographic embedding scheme, we submitted each repacked sample to the VT platform[6] using the code available as a module of Obfuscapk [4], a tool employed to obfuscate Android applications at different levels (e.g., ClassRename, and CallIndirection) and a enterprise API key. VT aggregates data from 79 antivirus engines and other cybersecurity services and sandboxes to detect malware or other malicious content, such as URLs and IP addresses.

After uploading the samples in VT, we retrieved the report listing all the information extracted from the application analysis. The most relevant data are *i*) static features gathered from the `AndroidManifest.xml` (i.e., permissions, activities and entry points), strings, used IPs and URLs, and APK meta information (i.e., signature, bundle file, file hashes); *ii*) the analysis of the various VT scanning tools to gather the number of detections the application had.

By analysing the reports, we looked at the number of tools that flagged a sample as malicious and the strings to understand whether the engines could retrieve the payloads that have been embedded through steganography within images and audio files.

5. Results

This section reports the outcome of our test campaign. First, it presents an analysis of the assets, and then it elaborates on the effectiveness of repackaging. Lastly, it discusses the performance of VT in terms of detection.

5.1. Asset Analysis

To generate the steganographic resources reported in Table 1, we first downloaded the top 20 real Android applications of the Google PlayStore in September 2024.

The images extracted from the applications are in PNG format, use RGBA or greyscale color models, and have three different resolutions (i.e., 32×32, 48×48, and 72×72 pixels). Instead, the audio files are in MP3 and WAV format, have an average duration of 2 seconds, and a sample rate of almost $44,000$ Hz. As a result, we obtained a dataset composed of 50 images hiding the portion of the `Necro` payload and 17 audio files hiding the list of URLs with the various steganographic methods and patterns described in Section 4.1. Again, we point out that not all the original resources were modified to hide content due to the constraints of the tools/techniques used to conduct experiments and model real-world stegomalware.

5.2. APK Repackaging

As mentioned, including steganography resources requires repackaging, rebuilding, and resigning the original application with the new images and audio. Therefore, to answer RQ1, we try to repack all the compromised applications. However, during the repackaging process, some applications could not be built successfully because of some APK protection mechanisms to avoid software cloning and cracking as well as to prevent the distribution of a malicious app mimicking its benign counterpart. In particular, `aapt`[7], a tool used by `ApkTool` to recompile the resources, raises an error (e.g., for Meta Inc. applications such as Facebook and Instagram). This also occurs in the case of unmodified applications,

[5]GitHub repo: https://github.com/Ocram95/AndroidStego/tree/main
[6]VirusTotal: https://www.virustotal.com/gui/home/upload
[7]aapt: https://developer.android.com/tools/aapt2

Application	LSB Plain	LSB Variant	AudioStego
Amazon-shopping	◐	-	●
BulletEcho	◐	-	-
CandyCrush	◐	◐	-
Contacts	◐	-	-
Duolingo	◐	-	●
Eurospin	◐	-	-
Facebook	◐	-	-
FlixBus	◐	-	-
Glovo	◐	◐	●
Idealista	◐◑	-	●
Instagram	◐	-	●
JustEat	◐	-	-
McDonalds	◐◑	◐◑	●
Messenger	◐	-	-
Revolut	◐	-	●
Shein	◐◑	◐◑	-
Telegram	◐	◐	●
TikTok	◐	◐	●
WhatsAppMessenger	◐	-	-
YouTube Music	◐	-	●

Table 1
Dataset of steganographic images and audio files hiding a payload. The ◐ and ◑ symbols denote images hiding the `Necro` payload via the sequential and the squares patterns, respectively. Instead, the ● symbol denotes audio files hiding the malicious URLs.

meaning that the included resources are not responsible for `aapt` error. In general, even though the repackaging is not working for some applications, this is good due to the tendency of attackers to apply the principle of *path of least resistance* [38]. This concept is based on the idea of exploiting vulnerabilities or writing malware employing the simplest techniques without necessarily focusing on a specific one.

Table 2 shows only the applications that were correctly repacked by including different combinations of the steganography resources, obtaining a final dataset of 67 new applications. For example, Amazon-Shopping has been repacked 3 times with 3 different combinations. After a successful repack, all of them were signed, with different development keys, and aligned as prescribed by Android documentation [39]. We also installed a random selection of 10 samples in an Android Emulator to check the functionalities preservation. All tested applications worked as the original version. To check the functionality, a manual analysis is needed, hence, we used a restricted number of apps. A more extensive investigation is part of our ongoing research.

5.3. Detection Effectiveness

To answer RQ2, we uploaded all the 67 steganographic apps in VT and automatically parsed the obtained reports. As reported in Table 2, only 2 applications were correctly detected by one out 79 antivirus. In particular, one sample for CandyCrush[8] and one for Duolingo[9] are detected by `VirIT` as `EICAR-Test-File` and `Trojan.Win32.AutoIt_Heur.L`, respectively. However, these results can be seen as false positives for two main reasons: *i*) the first is a test sample developed by EICAR [40], while the second is a Trojan for Windows 32 [41], which is not the target of an Android application; *ii*) we should consider the resulting label as a false positive if the detections by VirusTotal AVs are less than a threshold, typically set to 5 [42]. Additionally, we found that even though the original Duolingo[10] is flagged as benign, there is an IoC inside the dex file (`classes3.dex`). The same IoC is not found

[8]SHA256: 8efaab7e52c6c7bae6e928d3874bb1e38fa1c2de198f74f0fe71e8939be38f46
[9]SHA256: a6f3913327061c70a6cfcb602962bf75d901574801803478cd515020024d2a62
[10]SHA256: 418c7a88ca09dd8e5318c2b590f6f3d02c772ef06fd1976360b7f615f1ff7de1

Application	Repacked	Detections	
		Original	Repacked
Amazon-Shopping	3	0	0
Bullet Echo	3	0	0
CandyCrush	5	0	1
Duolingo	15	0(*)	1
Eurospin	3	0	0
FlixBus	3	0	0
Telegram	35	0	0

Table 2
Repacked applications and their detection results by 79 VT antivirus. Repacked is the number of successful repacked applications, while Detections denotes the AV detections for the Original application and the Repacked one. The (*) symbol denotes that despite VT detection being 0, some malicious IoCs have been detected via static analysis.

inside the malicious Duolingo app, meaning the application evaluation is negligible.

Analysis of the reports revealed that none of the hidden payloads were detected by VirusTotal AVs. This limitation arises because VT relies on `Androguard`[11], which focuses on static features extraction [43]. In our opinion, this is a strong limitation of the tool, which is not able to retrieve steganography resources. As a possible solution, an additional engine/layer could be implemented in VT for checking digital media against steganographic content. For example, this could be done by implementing the AI-based techniques described in [32].

6. Conclusion and Future work

In this paper, we presented an investigation of the "susceptibility" of Android applications against using steganography for creating advanced stealthy threats. To this aim, we conducted trials with 20 popular applications collected from the Google Play store, which have been repacked to contain image and audio files cloaking data through different steganographic mechanisms. Our tests indicated the feasibility of this threat model since none of our 67 repacked applications were flagged as malicious by antivirus engines interrogated by VirusTotal, with the exception of two detection instances that we deem as false positives. As a consequence, considering malware endowed with some form of information-hiding capabilities should be incorporated in the security assessment of future mobile ecosystems (e.g., applications and distribution stores) since the lack of additional controls may open up to a wide range of covert threats.

The limit of our investigation is that it is focused on manipulating pre-existent application assets without producing any alteration in terms of properties (e.g., the size of the file) or functional qualities (e.g., the perceived visual quality). For this reason, future works aim to broaden the considered research questions. Specifically, we want to evaluate whether the need to add a loader to unpack/decode the hidden data produces some signatures that can help to detect stegomalware. We are also working towards submitting applications to alternative stores to evaluate whether built-in checks may early prevent the use of information-hiding mechanisms.

Acknowledgments

This work was partially supported by Project SERICS (PE00000014) and Project RAISE (ECS00000035) under the NRRP MUR program funded by the EU - NGEU. This work was carried out while Silvia

[11]Androguard: https://androguard.readthedocs.io/en/latest/

Lucia Sanna was enrolled in the Italian National Doctorate on Artificial Intelligence run by Sapienza University of Rome in collaboration with the University of Cagliari.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] H. Haidros Rahima Manzil, S. Manohar Naik, Detection approaches for android malware: Taxonomy and review analysis, Expert Systems with Applications 238 (2024). doi:10.1016/j.eswa.2023.122255.

[2] H. Bostani, V. Moonsamy, EvadeDroid: A practical evasion attack on machine learning for black-box Android malware detection, Computers & Security 139 (2024) 103676. doi:https://doi.org/10.1016/j.cose.2023.103676.

[3] F. Pierazzi, F. Pendlebury, J. Cortellazzi, L. Cavallaro, Intriguing Properties of Adversarial ML Attacks in the Problem Space, in: 2020 IEEE Symposium on Security and Privacy (SP), 2020, pp. 1332–1349. doi:10.1109/SP40000.2020.00073.

[4] S. Aonzo, G. C. Georgiu, L. Verderame, A. Merlo, Obfuscapk: An open-source black-box obfuscation tool for Android apps, SoftwareX 11 (2020) 100403. doi:https://doi.org/10.1016/j.softx.2020.100403.

[5] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, F. Roli, Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection, IEEE Transactions on Dependable and Secure Computing 16 (2019) 711–724. URL: https://doi.org/10.1109/TDSC.2017.2700270.

[6] G. Suarez-Tangil, J. Tapiador, P. Peris-Lopez, Stegomalware: Playing Hide and Seek With Malicious Components in Smartphone Apps, in: International Conference on Information Security and Cryptology, 2014, pp. 496–515.

[7] S. Tanner, I. Vogels, R. Wattenhofer, Protecting Android Apps from Repackaging Using Native Code, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 12056 LNCS (2020) 189 – 204. doi:10.1007/978-3-030-45371-8_12.

[8] T. Tu, H. Zhang, Y. Hu, D. Cui, X. Zhai, A detector for Android repackaged applications with layout-fingerprint, Journal of Information Security and Applications 76 (2023) 103527. doi:https://doi.org/10.1016/j.jisa.2023.103527.

[9] Necro Trojan attack, https://www.kaspersky.com/blog/necro-infects-android-users/52201/, 2024. Accessed: 11/2024.

[10] L. Caviglione, W. Mazurczyk, Never Mind the Malware, Here's the Stegomalware, IEEE Security & Privacy 20 (2022) 101–106.

[11] F. Strachanski, D. Petrov, T. Schmidbauer, S. Wendzel, A Comprehensive Pattern-based Overview of Stegomalware, in: Proceedings of the 19th International Conference on Availability, Reliability and Security, 2024, pp. 1–10.

[12] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket, 2014. doi:10.14722/ndss.2014.23247.

[13] Y. Pan, X. Ge, C. Fang, Y. Fan, A Systematic Literature Review of Android Malware Detection Using Static Analysis, IEEE Access 8 (2020) 116363 – 116379. doi:10.1109/ACCESS.2020.3002842.

[14] A. Martín, V. Rodríguez-Fernández, D. Camacho, CANDYMAN: Classifying Android malware families by modelling dynamic traces with Markov chains, Engineering Applications of Artificial Intelligence 74 (2018) 121–133. doi:https://doi.org/10.1016/j.engappai.2018.06.006.

[15] S. Arshad, M. A. Shah, A. Wahid, A. Mehmood, H. Song, H. Yu, SAMADroid: A Novel 3-Level

Hybrid Malware Detection Model for Android Operating System, IEEE Access 6 (2018) 4321 – 4339. doi:10.1109/ACCESS.2018.2792941.

[16] A. Ruggia, D. Nisi, S. Dambra, A. Merlo, D. Balzarotti, S. Aonzo, Unmasking the Veiled: A Comprehensive Analysis of Android Evasive Malware, in: Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS '24, Association for Computing Machinery, 2024, p. 383–398. doi:10.1145/3634737.3637658.

[17] V. Sihag, A. Swami, M. Vardhan, P. Singh, Signature Based Malicious Behavior Detection in Android, in: Computing Science, Communication and Security, Springer Singapore, 2020, pp. 251–262.

[18] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, G. Stringhini, MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version) 22 (2019). doi:10.1145/3313391.

[19] V. Malhotra, K. Potika, M. Stamp, A comparison of graph neural networks for malware classification, Journal of Computer Virology and Hacking Techniques 20 (2024) 53 – 69. doi:10.1007/s11416-023-00493-y.

[20] S. HR, Static Analysis of Android Malware Detection using Deep Learning, in: 2019 International Conference on Intelligent Computing and Control Systems (ICCS), 2019, pp. 841–845. doi:10.1109/ICCS45141.2019.9065765.

[21] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, L. Cavallaro, DroidScribe: Classifying Android Malware Based on Runtime Behavior, in: 2016 IEEE Security and Privacy Workshops (SPW), 2016, pp. 252–261. doi:10.1109/SPW.2016.25.

[22] Z. Yuan, Y. Lu, Y. Xue, Droiddetector: Android malware characterization and detection using deep learning, Tsinghua Science and Technology 21 (2016) 114–123.

[23] D. Li, Q. Li, Adversarial Deep Ensemble: Evasion Attacks and Defenses for Malware Detection, IEEE Transactions on Information Forensics and Security 15 (2020) 3886 – 3900. doi:10.1109/TIFS.2020.3003571.

[24] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, K. Ren, Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection, IEEE Transactions on Information Forensics and Security 15 (2020) 987 – 1001. doi:10.1109/TIFS.2019.2932228.

[25] J. Guo, D. Liu, R. Zhao, Z. Li, Wltdroid: Repackaging detection approach for android applications, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 12432 LNCS (2020) 579 – 591. doi:10.1007/978-3-030-60029-7_52.

[26] L. Caviglione, M. Gaggero, J.-F. Lalande, W. Mazurczyk, M. Urbański, Seeing the unseen: revealing mobile malware hidden communications via energy consumption and artificial intelligence, IEEE Transactions on Information Forensics and Security 11 (2015) 799–810.

[27] D. Dell'Orco, G. Bernardinetti, G. Bianchi, A. Merlo, A. Pellegrini, Would You Mind Hiding My Malware? Building Malicious Android Apps with Stegopack, SSRN (2024). URL: https://ssrn.com/abstract=5039499.

[28] J. Blasco, J. Hernandez-Castro, J. M. de Fuentes, B. Ramos, A Framework for Avoiding Steganography Usage Over HTTP, Journal of Network and Computer Applications 35 (2012) 491–501.

[29] A. Schaffhauser, W. Mazurczyk, L. Caviglione, M. Zuppelli, J. Hernandez-Castro, Efficient Detection and Recovery of Malicious PowerShell Scripts Embedded into Digital Images, Security and Communication Networks 2022 (2022) 4477317.

[30] D. Puchalski, L. Caviglione, R. Kozik, A. Marzecki, S. Krawczyk, M. Choraś, Stegomalware Detection Through Structural Analysis of Media Files, in: Proceedings of the 15th International Conference on Availability, Reliability and Security, 2020, pp. 1–6.

[31] R. Han, C. Yang, J. Ma, S. Ma, Y. Wang, F. Li, IMShell-Dec: Pay More Attention to External Links in PowerShell, in: SEC, volume 580 of *IFIP Advances in Information and Communication Technology*, Springer, 2020, pp. 189–202.

[32] M. Guarascio, M. Zuppelli, N. Cassavia, L. Caviglione, G. Manco, Revealing MageCart-like Threats in Favicons via Artificial Intelligence, in: ARES, ACM, 2022, pp. 45:1–45:7.

[33] S. Geetha, S. Subburam, S. Selvakumar, S. Kadry, R. Damasevicius, Steganogram removal using multidirectional diffusion in Fourier domain while preserving perceptual image quality, Pattern Recognition Letters 147 (2021) 197–205.

[34] H. Tao, L. Chongmin, J. M. Zain, A. N. Abdalla, Robust Image Watermarking Theories and Techniques: A review, Journal of Applied Research and Technology 12 (2014) 122–138.

[35] P. K. Robinette, H. D. Wang, N. Shehadeh, D. Moyer, T. T. Johnson, SUDS: Sanitizing Universal and Dependent Steganography, in: European Conference on Artificial Intelligence, volume 372, 2023.

[36] M. Zuppelli, G. Manco, L. Caviglione, M. Guarascio, Sanitization of Images Containing Stegomalware via Machine Learning Approaches, in: Proceedings of the Italian Conference on Cybersecurity, volume 2940, 2021, pp. 374–386.

[37] A. Zimba, Z. Wang, H. Chen, Multi-stage Crypto Ransomware Attacks: A new Emerging Cyber Threat to Critical Infrastructure and Industrial Control Systems, ICT Express 4 (2018) 14–18.

[38] B. J. Sagarin, K. D. Mitnick, 27the path of least resistance, in: Six Degrees of Social Influence: Science, Application, and the Psychology of Robert Cialdini, Oxford University Press, 2012. URL: https://doi.org/10.1093/acprof:osobl/9780199743056.003.0003.

[39] Build you app from comand line, https://developer.android.com/build/building-cmdline?hl=en, 2024. Accessed: 11/2024.

[40] What is the EICAR test file?, https://www.eicar.org/download-anti-malware-testfile/, 2024. Accessed: 11/2024.

[41] Trojan.Win32.Autoit, https://threats.kaspersky.com/en/threat/Trojan.Win32.Autoit/, 2024. Accessed: 11/2024.

[42] S. Zhu, J. Shi, L. Yang, B. Qin, Z. Zhang, L. Song, G. Wang, Measuring and Modeling the Label Dynamics of Online Anti-Malware Engines, in: 29th USENIX Security Symposium (USENIX Security 20), USENIX Association, 2020, pp. 2361–2378.

[43] Androguard VirusTotal, https://docs.virustotal.com/reference/androguard, 2024. Accessed: 11/2024.