Challenging Antivirus against Elusive Android Malware over Time

Danilo Dell'Orco[1], Lorenzo Valeriani[1,2], Giuseppe Bianchi[1,2], Alessandro Pellegrini[2] and Alessio Merlo[3,*]

[1]CNIT National Network Assurance and Monitoring (NAM) Lab, Rome, Italy
[2]University of Rome Tor Vergata, Rome, Italy
[3]CASD - School of Advanced Defense Studies, Rome, Italy

Abstract

This empirical study evaluates the ability of commercial Android antivirus (AV) solutions to detect malware concealed using four specific techniques: code obfuscation and repackaging through encryption, compression, and steganography. Using real-world malware samples spanning 14 years, we developed a structured testing pipeline to transform each sample into a less-detectable version systematically. Leading antivirus engines on VirusTotal, which reliably detect the original malware, were tested against the altered malware. Although the effectiveness of the repackaging techniques varies, they all significantly reduce detection rates. While obfuscation reduces detection rates, most AV engines remain resilient to state-of-the-art obfuscation tools. We conducted the same tests at five intervals, demonstrating that AV systems quickly adapt to these hiding techniques. However, this adaptation often results in inadequate signatures. In some cases, engines misclassified the original host application as malicious, even without a hidden payload. While VirusTotal remains a valuable resource for malware analysis, our experiments highlight several limitations. These findings underscore the need for more robust threat-neutralizing techniques and advanced detection strategies to address the evolving challenges of malware in Android ecosystems. To improve the reliability of such studies, we propose best practices for conducting experiments.

Keywords

Stegomalware, Android Security, Packing, Obfuscation, Antivirus Evasion

1. Introduction

Android, built on the Linux kernel, is often regarded as a secure operating system, with foundational security measures such as explicit permission granting, application isolation, mandatory app signing, and a disabled root user. These fundamental mechanisms establish a strong security framework. However, as the threat landscape evolves, Android faces increasingly sophisticated attacks targeting this ecosystem [1, 2].

Unlike traditional Linux or Windows malware, which typically seeks deep OS-level access, Android malware frequently operates within the app sandbox. It leverages the permissions framework to carry out malicious actions, blurring the line between legitimate and malicious applications. While commercial antivirus solutions primarily rely on signature-based detection, identifying malware only after its signature has been analyzed and cataloged, some solutions, like Google Play Protect, also incorporate advanced techniques such as code analysis and machine learning to detect threats more proactively and reduce the reliance on known signatures [3, 4].

Joint National Conference on Cybersecurity (ITASEC & SERICS 2025), February 03-8, 2025, Bologna, IT
*Corresponding author.
danilo.dellorco@cnit.it (D. Dell'Orco); lorenzo.valeriani@cnit.it (L. Valeriani); giuseppe.bianchi@uniroma2.it (G. Bianchi); alessandro.pellegrini@uniroma2.it (A. Pellegrini); alessio.merlo@unicasd.it (A. Merlo)
http://netgroup.uniroma2.it/GiuseppeBianchi/biografia.html (G. Bianchi); http://netgroup.uniroma2.it/GiuseppeBianchi/biografia.html (A. Pellegrini); https://www.csec.it/people/alessio_merlo (A. Merlo)
0009-0004-3539-7236 (D. Dell'Orco); 0009-0003-9373-9575 (L. Valeriani); 0000-0001-7277-7423 (G. Bianchi); 0000-0002-0179-9868 (A. Pellegrini); 0000-0002-2272-2376 (A. Merlo)

In response, malware developers constantly refine *hiding techniques* to avoid detection and target more victims. These techniques are designed to disrupt signature-based detection and machine learning models' feature extraction process, necessitating increasingly sophisticated countermeasures.

One common strategy is *app repackaging*, where attackers inject malicious payloads into legitimate apps before redistributing them. Standard hiding techniques include *code obfuscation* and *encryption*. Obfuscation restructures the code to make analysis more difficult while retaining its functionality [5]. Tools such as ObfuscAPK [6] and ProGuard [7] effectively bypass many detection systems. Commercial packers [8, 9, 10] use encryption to protect the application code, further complicating the analysis.

On the other hand, steganography represents a more sophisticated and elusive technique [11]. By embedding malicious payloads within non-code resources such as images or audio files, steganography enables the creation of *stegomalware* [12]. Stegomalware is increasingly used on Android to bypass detection mechanisms [13, 14, 15]. This rise can be attributed to malicious payloads being hidden from static analysis by revealing them only at runtime. Steganography does not alter the app's entropy, unlike encryption [16], making it harder for antivirus systems to detect hidden threats without executing the app.

This study assesses the effectiveness of various malware-hiding techniques, explicitly focusing on obfuscation and different repackaging strategies. To evaluate how well antivirus solutions are prepared for repackaging, we analyze several techniques of varying complexity: a *standard encryption scheme*, the more straightforward method of *compression*, and the more covert use of *steganography*.

This research aims to address the following key questions:

- **RQ1**: How do Android antivirus solutions respond to different malware hiding techniques?
- **RQ2**: How does the year of malware creation influence its detection rate?
- **RQ3**: How do antivirus engines evolve to counteract these threats?

Our findings highlight significant weaknesses in existing antivirus solutions for Android, particularly in detecting advanced malware-hiding techniques. We observed that traditional signature-based detection methods struggle against complex repackaging strategies such as encryption and steganography-based hiding, completely bypassing static analysis. Furthermore, the evolution of malware demonstrates that detection engines must constantly adapt to new techniques, mainly when obfuscation and steganography are increasingly used. These results underscore the need for more advanced and dynamic detection approaches to address the growing sophistication of Android malware.

The remainder of the paper is structured as follows. Section 2 provides information on Android and related work. Section 3 outlines our methodology. Section 4 presents the experimental results, and Section 5 concludes the study.

2. Background and Related Work

2.1. Android Basics

The Android platform is an open-source software stack built on the Linux kernel. It is suitable for various devices. At its core, the kernel manages system resources, including memory, processes, and device drivers, supporting a robust environment for efficient Android application execution. The Hardware Abstraction Layer (HAL) facilitates standardized interfaces for hardware capabilities like Bluetooth or Camera modules and integrates seamlessly with the higher-level Java API framework.

Each Android application runs within a dedicated Android RunTime (ART) instance using a distinct Linux user ID (UID). This sandboxed design ensures isolated memory spaces and prevents direct data access between applications. Applications are written in Kotlin/Java code, organized in a set of independent *components* belonging to four categories: *Activities*, representing the app's graphical UIs; *Services*, executing background-running tasks without any user's interaction; *Content Providers*, allowing the app to export part of its database to other apps; and *Broadcast Receivers*, consisting of tasks executed when specific events occur.

Compiled apps are installed as APK files, i.e., a compressed file format signed by the developer containing code and non-code resources. The main elements of an APK are *dex files*, *resources*, *assets*, *shared libraries*, and the manifest file. The dex files (i.e., `classes[k].dex`) contain the compiled code (bytecode) of the Java/Kotlin classes, which represent the core functionality of the app. Resources are compiled entities, stored in the `res/` folder and indexed in the `resources.arsc` file. Assets are non-compiled resources in the `assets/` folder. They can cover various file types, including fonts, videos, and audio files. Unlike resources in the `res/raw/` subfolder, which are compiled into the application and referenced via resource IDs, assets are accessed directly in their original form by specifying their file path. Shared libraries, which are stored in the `lib/` folder, consist of ELF binaries (*.so* files) built by compiling architecture-specific native code (C/C++). The `AndroidManifest.xml` file declares the set of the app's components and the set of permissions that the app aims to exploit at runtime.

2.2. App Modification and Hiding Techniques

APK files can be reversed using decompiling tools (e.g., ApkTool [17]) that provide access to original resources and convert the app into *smali*, a human-readable assembly-like language representing the Dalvik bytecode of the app. The inherent reversibility of Android apps [18] has also encouraged the development of techniques to hinder reverse engineering and malware analysis.

Obfuscation. Obfuscation refers to the practice of making code difficult to understand, both for reverse engineers and automated analysis tools [5]. In Android, benign apps widely use obfuscation to protect intellectual property, while malware uses it to hide malicious payloads. Tools such as ProGuard [7] or custom obfuscators transform the code by renaming methods and variables, adding dead code, or altering the control flow.

Zheng et al.[19] demonstrated that automated obfuscation significantly reduced antivirus detection rates in the short term. However, after four months, the detection rates increased from 54% to 90%, as the antivirus engines adapted to these techniques. Several studies [20, 21, 22, 23] have proposed solutions for detecting obfuscated malware, including tools like APKiD [24], which uses Yara rules to identify known obfuscators.

Repackaging. Repackaging involves modifying an app's compiled APK by inserting new code or altering existing components, allowing an attacker to insert malicious code, recompile the app, and redistribute it [25, 26, 27, 28, 29]. If the malicious code is kept in plain sight, it is easily detectable by static analysis tools. Repackaging is often combined with techniques that hide the malicious payload and evade static detection to avoid this.

Commercial packers, such as Baidu [10], Bangcle [9], and Ijiami [8], automate this process to protect apps while simultaneously making reverse engineering and analysis significantly harder. These packers often encrypt the original app's `classes.dex`, embedding it within the APK and decrypting it at runtime using APIs from the `DexClassLoader` family. Tools such as DexHunter [30], AppSpear [31], and BadUnboxing [32] have been developed to extract original code from packaged apps, although they face challenges against custom packagers or layered protection techniques.

Encryption. Encryption-based repackaging [33] encrypts the app's `classes.dex` to prevent static analysis. The code is decrypted and executed dynamically at runtime, bypassing static detection by antivirus tools. Although this approach increases complexity for reverse engineers, it also introduces runtime overhead and is susceptible to dynamic analysis tools that can intercept the decryption process.

Compression. Compression-based repackaging [34] is more straightforward compared to encryption. It involves compressing the components of the app to obscure its content. However, unlike encryption, compression is not designed to hide information. As a result, compressed apps can be easily decompressed by antivirus engines or reverse engineering tools, revealing the original app's content without executing it.

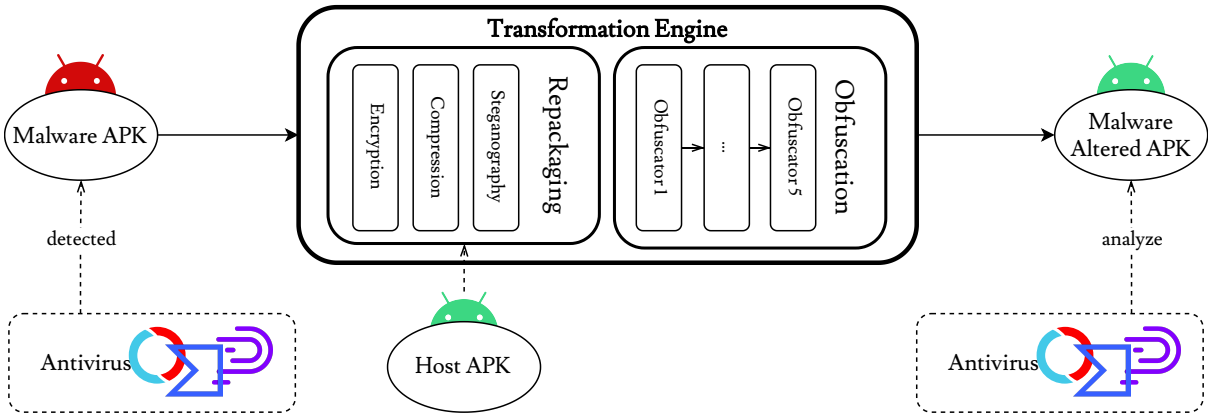


Figure 1: Malware Transformation Pipeline.

Steganography. Steganography refers to techniques to embed information in communication channels or files, such as images or audio. A standard method is the *Least Significant Bit* (LSB) scheme [35], which alters the least significant bit of pixel values to encode hidden data, exploiting the imperceptibility of small changes in the color channels. In addition to being used for legitimate purposes (e.g., discreet data transmission), steganography can be exploited in stegomalware [12] to conceal code, deliver payload, and establish covert command and control channels.

Android malware increasingly exploits steganography to evade detection, revealing significant limitations in antivirus defenses. Badhani et al.[36] showed that hiding malware in images bypassed detection by nine out of ten antivirus tools. Suárez-Tangil et al. [37] emphasized the challenge of statically identifying stegomalware, as many legitimate applications perform image manipulation operations similar to steganographic decoding. Our previous work, Stegopack [11], further revealed that antivirus systems are ineffective against stegomalware. Although the proposed solution combines steganalysis with dynamic analysis to successfully detect stegomalware, developing a more general detection mechanism independent of the specific embe dding scheme is not trivial.

3. Approach and Methodology

Our approach, shown in Fig. 1, is built around a modular tool designed to implement and evaluate four distinct malware hiding techniques: code obfuscation, naive repackaging using compression, repackaging using encryption, and repackaging with steganography. We applied such techniques to existing malware APKs to assess their effectiveness against detection systems. Each method is independently configurable within the tool, allowing for precise experimentation and comparing their impact on antivirus detection rates.

3.1. Obfuscation

To obfuscate malware, we utilize Obfuscapk [6], a tool that enables black-box obfuscation of Android applications without requiring access to the source code. The obfuscation process involves several techniques, such as encrypting resources, altering control flow, and using reflection to invoke methods dynamically. The APK is first decompiled using Apktool, which exposes its smali code. The obfuscators then modify the smali to alter its structure while preserving the malware's functionalities. Finally, the application is recompiled and signed, resulting in a functional APK with a modified signature and obfuscated code. For the obfuscation of Java-based malware, we applied a combination of five obfuscation techniques known for their effectiveness in significantly scrambling the code structure [38]: `ArithmeticBranch`, `Reorder`, `MethodOverload`, `CallIndirection`, and `Goto`. For malware containing native code, we added an obfuscator (`LibEncryption`) to encrypt the `so` files.

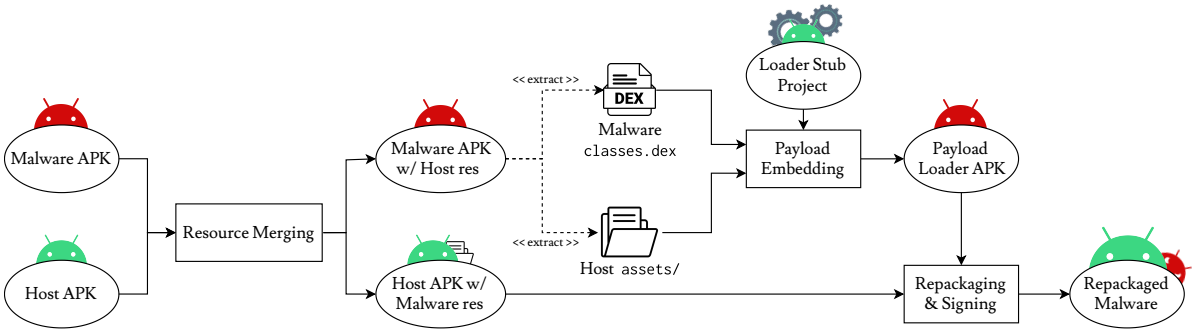


Figure 2: Malware Repackaging Workflow.

3.2. Repackaging

To evaluate the impact of different repackaging techniques, we built a structured pipeline to automate the process, shown in Fig. 2. The process starts by decompiling the host and malware APKs using Apktool.

The *Resource Merger* integrates the malware resources into the host app, following the approach proposed in [11]. This involves renaming the resources to avoid conflicts and resolving resource ID overlaps by increasing unique identifiers. Consistency is maintained by updating all affected files, including XML and smali code, with minimal overhead.

Next, raw payload components are extracted directly from the APK without decompiling. These include the `classes.dex` file and the `lib/` folder containing native libraries. The payload files are then embedded into the host app's assets using the *Payload Loader*, applying Compression, Encryption, or Steganography. These encoded assets are integrated into the Loader Stub project, and the *loader APK* is built using the `gradlew` command.

The *Repackager* inserts the *Payload Loader* into the host APK. This step adds the loader classes, updating the `AndroidManifest.xml` to include the permissions required by the payload and configuring the host to invoke the loader at specific points to dispatch the malware. At the end of the process, the final APK remains functional and retains the appearance of the original host app.

3.2.1. Payload Loader

The Payload Loader is a pivotal intermediary that links the host app and the hidden malicious payload. It performs two essential functionalities during the execution time, namely *Payload Extraction* and *Runtime Malware Execution*.

First, the loader reverses the algorithm used in the embedding phase (steganography, compression, or encryption) to extract from the host's assets the original bytes of both Java and native payload. The `classes.dex` bytes are stored in a `ByteBuffer`, that is passed to the constructor of the `InMemoryDexClassLoader` [39]. This *ClassLoader* allows dynamically loading DEX files not initially included in the host APK. Using Java Reflection, the custom class loader is set as the active one for the host app, enabling the load of host and payload classes. The malicious class is dynamically loaded via the `loadClass()` method, followed by starting its associated activity using the `startActivity()` API.

The native libraries are hidden using the same approach proposed in Stegopack [11]: the Payload Loader extracts the bytes of each native library from its respective source (image, compressed file, encrypted file) and writes them to corresponding `.so` files within the host's private directory under `/data/data`. To enable the dispatched payload to load native libraries, we specify as `librarySearchPath` parameter the path `/data/data/com.host.app`. This way, native libraries extracted to that directory can be located and loaded using the `System.loadLibrary()` method

during runtime.

For payload embedding, we use established techniques to hide `classes.dex` and `.so` files. A modified *Least Significant Bit* (LSB) algorithm [40] is used for steganography to embed the payload within the host's image assets. For encryption, the *Advanced Encryption Standard* (AES) is applied, and for compression, *Zstandard* with a compression level of 3 is used. The `classes.dex` file and the packet containing all the `.so` files are stored in their encrypted or compressed form as additional assets within the host's resources.

3.2.2. Host App Modification

We utilize a repackaging attack to integrate malware functionality into the host app. We inject only the steganographic loader app to minimize the customization of the host app by incorporating just a single additional activity. We modify the host's *main activity* by customizing the implementation of the `onCreate()` method in the corresponding `smali` file. Here, we insert an `invoke-static` call to activate the steganographic loader, which decodes and executes malware. Once the loader executes the payload as a standard activity within the host app's context, the runtime behavior is the same as the original app, containing only the permissions the host requires. To ensure the functionality of the payload, we extract the permission needed from the malware manifest during repackaging and attach them to the appropriate `intent-filter` elements in the host's manifest.

4. Experiments

Our dataset contains 140 malware samples from *VX Underground*, *github.com/sk3ptre*, and *Malware Bazaar* spanning 14 years. We selected ten distinct malware families each year, evenly divided between Java-only and native. We applied four hiding techniques, namely *obfuscation*, *compression*, *encryption*, and *steganography*, to each malware sample, thus obtaining a dataset of 700 malware samples.

To assess the effectiveness of these techniques, we submitted all 700 samples to VirusTotal [41] and analyzed the detection results from 79 antivirus engines. This evaluation allowed for the measurement of the impact of each hiding method on malware detection rates and the exploration of trends in antivirus capabilities over time. For repackaging techniques, we utilized as a goodware host a simple number guessing game application not known to VirusTotal.

We compare the total detection of the different techniques in Fig. 3. Detailed data is in Table 1. The techniques showing the highest hiding result are steganography, encryption, and compression. Obfuscation is less effective than the other approaches. The results in Table 1 also highlight the difference in the presence of malware detected on Java and native applications. Java shows slightly higher detection rates for most categories, indicating marginally worse performance in resisting malware detection than native. This fact may stem from malware developers often targeting Java code, which is easier to analyze and manipulate than native binaries.

However, in terms of obfuscation, native malware performs worse, with more threats detected (1088 vs. 930). This anomaly could be attributed to AV engines leveraging tools like `APKiD` [24], which includes specific YARA rules targeting obfuscated native libraries, such as `LibEncryption` [42]. Since some AV engines may embed similar techniques, native libraries may be more susceptible to detection in this case.

Figures 4, 5, 6, 7 shows the ability of AVs to classify modified malware under its original threat category[1]. Notably, with compression, AVG (76/140), Kaspersky (102/140), ESET (125/140), and Avast (76/140) detect most of the original threats. This suggests that these antiviruses, historically prominent on desktop platforms, effectively handle compressed payloads by decompressing malware and applying signature matching to the extracted `dex` or `so` files.

[1] By *"false positive"*, we mean malware initially undetected but flagged as malicious after applying the hiding technique. *"Different detection"* refers to malware identified as a different threat than its original classification.

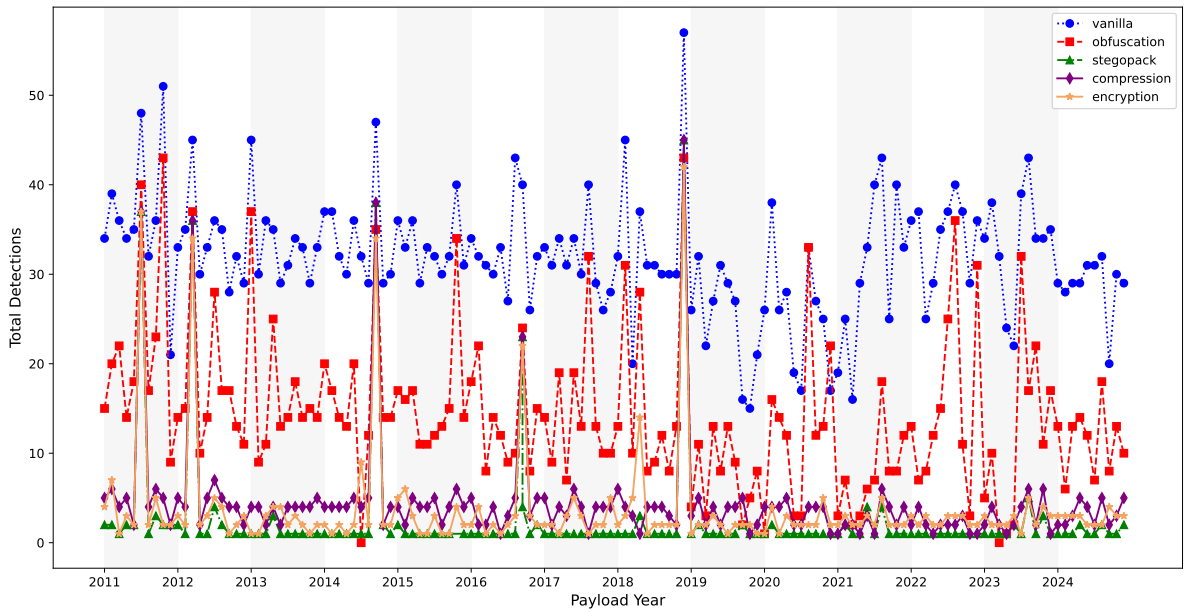


Figure 3: Total detection comparison of different hiding techniques through the years.

Regarding obfuscation, if a threat is detected, it is consistently classified under the original category. This technique also introduces 156 false positives out of 11,060 samples, indicating that applications previously undetected by a specific antivirus (AV) are flagged as threats after obfuscation. In comparison, other techniques result in fewer false positives: compression introduces 28, while encryption and steganography introduce 24.

ESET consistently flagged all malware samples as malicious across the three repackaging techniques. It successfully identified 125 out of 140 malware samples in the case of compression. However, for both steganography and encryption, it classified the malware as `Android/TrojanDropper.Agent.MHY` in 102 instances. As discussed in [11], these detections are specific to the Payload Loader class, regardless of whether the loaded class is malicious. Previous studies have shown that this detection can be easily bypassed by renaming project elements (such as assets, packages, classes, methods, and variables) and applying simple code-scrambling techniques to the payload loader. Thus, this detection can also be considered a false positive, as it is independent of the repackaged application and can even flag a goodware payload.

Fig. 8 illustrates the detection performance of different antivirus (AV) vendors against the different hiding techniques. The results indicate a clear distinction between the detection capabilities. Although some AV engines perform strongly against compression-based hiding, overall detection rates for repackaging techniques are generally low. Among these, steganography shows slightly better detection rates than encryption, which outperforms compression. However, all AVs demonstrate consistent preparedness against obfuscation, achieving significantly higher detection rates than other techniques.

Table 1
Detection rates for each hiding technique.

Hiding Technique	Total Detections	Java	Native	Detection Rate (%)
Vanilla	4464	2275	2189	40.36
Obfuscation	2018	930	1088	18.24
Compression	642	368	274	5.79
Encryption	521	284	237	3.40
Steganography	357	209	148	2.01

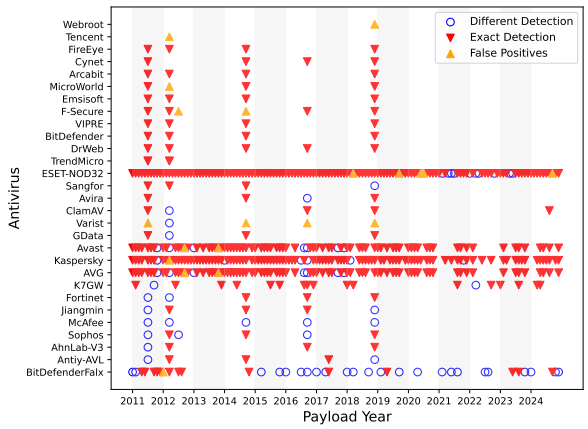


Figure 4: AV threat detection after compression.

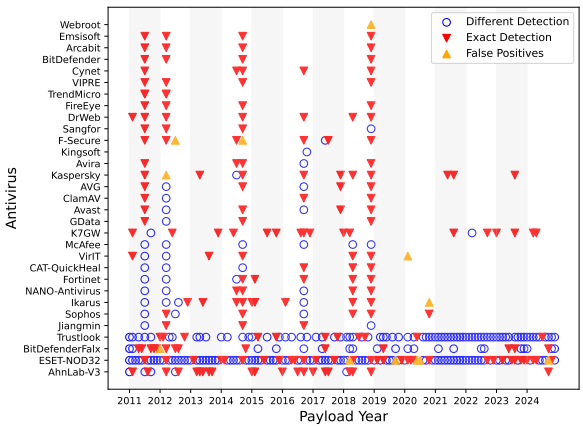


Figure 5: AV threat detection after encryption.

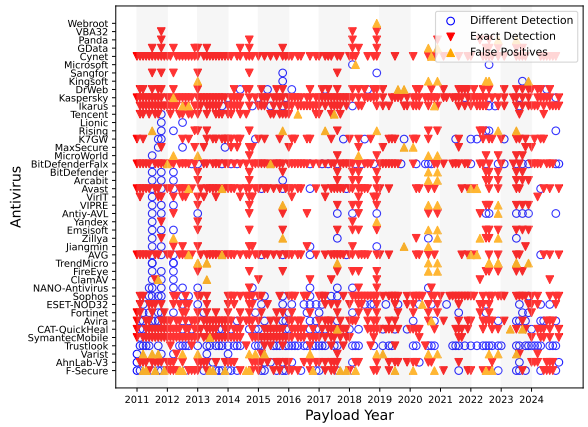


Figure 6: AV threat detection after obfuscation.

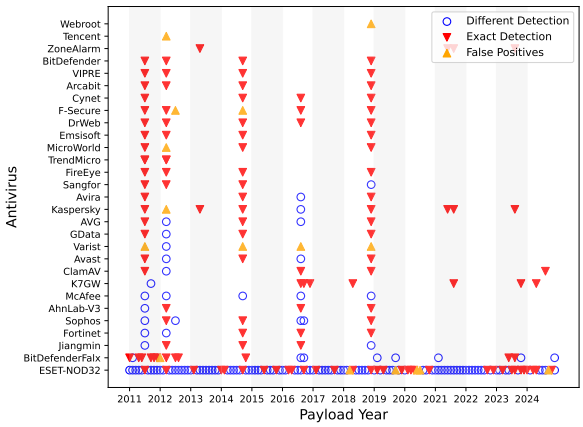


Figure 7: AV threat detection after steganography.

4.1. Repeating Experiments

We repeated the experiments every two days to assess how quickly antivirus software adapts to different hiding techniques. As shown in Fig. 9, antivirus detection rates rose significantly against obfuscation, with detections increasing from 2,018 on the first day to 2,672 on the last. Against compression AVs performed even better, with detections tripling from 642 to 2,032 over the same period. In contrast, steganography and encryption showed more consistent detection trends. Steganography stabilized around 694 detections, while encryption gradually increased from 521 to 818 detections.

A reanalysis of the goodware host application on VirusTotal revealed that four antivirus engines (Avira, Cynet, Avast-Mobile, and ESET) identified it as malicious. However, resigning the original app with a fresh keystore reduced detections to just two, suggesting that specific antivirus heuristics were based on the developer's signing certificate rather than detecting malicious content. To validate this hypothesis, we signed a previously undetected application with the same compromised keystore used in the experiments, which subsequently caused it to be flagged as malicious.

Nonetheless, two detections persisted regardless of the keystore, indicating that the original goodware application was classified as malicious solely for being used in malware repackaging. This behavior underscores a critical issue: while detection rates may improve, *antivirus engines often generate signatures based on incorrect aspects of the application*, such as the benign goodware component. This flaw exposes applications to reputation attacks, where an attacker can repeatedly embed malware samples into any target app and submit it to VirusTotal, intentionally triggering false positives for legitimate applications.

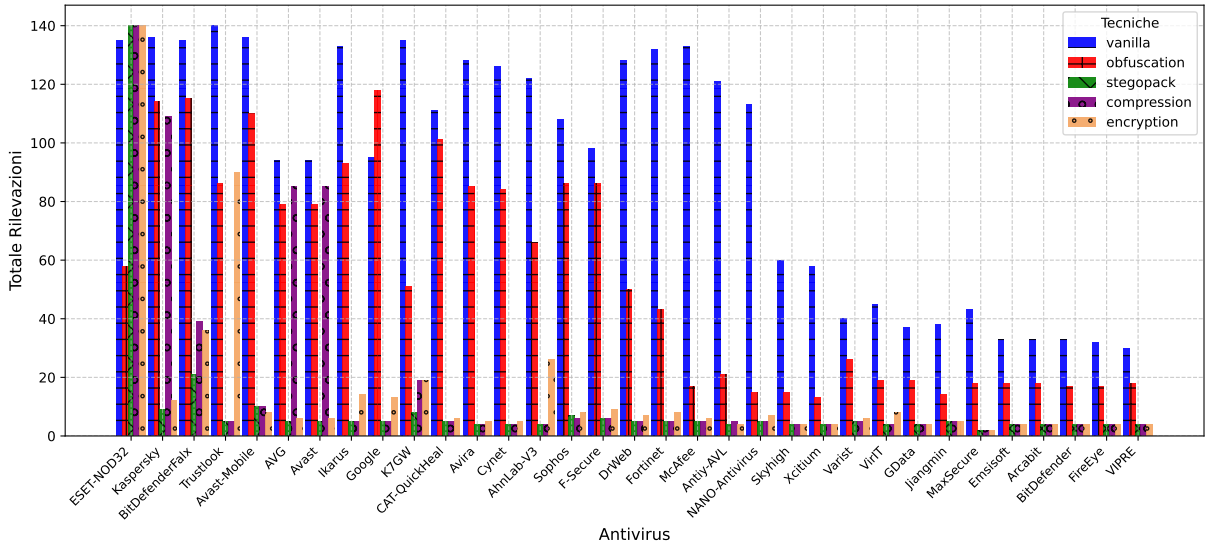


Figure 8: Antivirus ranking against different hiding techniques.

4.2. Discussion

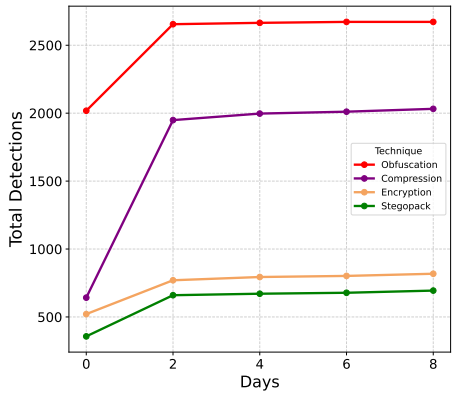
From the experiments carried out, it can be observed that the year of malware creation does not significantly influence its detection rates. This phenomenon can be attributed to the dataset creation process. Specifically, we labeled an application as malware if more than 15 antivirus engines flagged it as malicious [11]. Consequently, the dataset consists of samples that are well-recognized by antivirus solutions. As demonstrated in Subsect. 4.1, antivirus engines quickly adapt to emerging threats. A year, therefore, represents a sufficiently long period for these engines to incorporate effective detection mechanisms.

However, it is reasonable to hypothesize that some of the more recent malware samples may not yet be widely recognized, potentially falling below the detection threshold. Excluding cases of misclassification, the results show that obfuscation reliably reduces detection rates. However, obfuscation is outperformed by all other evasion techniques tested. Its diminished effectiveness may be related to the fact that the obfuscation tool used represents state-of-the-art technology that has been well studied, allowing antivirus engines ample time to develop robust countermeasures.

Other techniques require a deeper discussion. Regarding compression, four antivirus engines demonstrated the ability to recognize the original threat in most cases. Excluding these four, compression's performance aligns closely with that of encryption. This finding is noteworthy because it underscores how the majority antivirus systems struggle to identify a basic form of repackaging, one that employs a standard lossless compression algorithm without specific information hiding purposes. Among the three techniques tested, steganography, particularly the one implemented in Stegopack, yields the most effective evasion results. By embedding the payload within existing assets, it minimizes compromise indicators in the APK. In contrast, compression and encryption introduce files with different formats into the APK, which can raise flags during analysis.

The rapid evolution of antivirus detections over a short period implies that experiments conducted via VirusTotal are not reproducible. To improve accuracy and ensure reproducibility, some strategies must be employed. First, it is essential to concentrate all experiments within the shortest possible time window to prevent later experiments from being biased by the knowledge acquired by AVs during earlier tests.

Moreover, we have demonstrated that, after multiple malware submissions, some AVs generate specific signatures targeting the loader, the host application, or the developer's signature. Therefore, to evaluate antivirus performance against new threats effectively, it is crucial to refactor the loader (or equivalent components) and apply a new developer signature between consecutive experiments.



Technique	Day 0	Day 2	Day 4	Day 6	Day 8
Vanilla	4464	4464	4464	4464	4464
Obfuscation	2018	2655	2665	2672	2672
Compression	642	1949	1997	2011	2032
Encryption	521	770	794	802	818
Steganography	357	660	671	678	694

Figure 9: AV total detection evolution over time in December 2024, presented both graphically (left) and in tabular form (right).

5. Conclusion

This study underscores critical gaps in the ability of Android antivirus solutions to detect advanced malware-hiding techniques. While naive obfuscation has seen diminishing effectiveness due to improved countermeasures, more sophisticated methods like steganography consistently bypass detection. By embedding malicious payloads within existing app assets, steganography avoids entropy changes and evades static analysis. Compression, a simple and lossless transformation, remains effective at concealing threats. It was found out that only 4 out of 79 antivirus engines were capable of readily detecting it. Although detection rates improved after several days, the delayed response is a cause of concern.

Our experiments reveal the need for antivirus solutions to evolve beyond static, signature-based methods, which struggle against rapidly evolving evasion strategies. The tendency to misclassify benign components, such as loaders or repackaged host applications, exposes a serious flaw. This not only reduces detection accuracy but also poses a reputation risk to legitimate applications unintentionally flagged as malicious.

Dynamic, behavior-based analysis must become central to malware defense strategies to enhance detection robustness. Future research should focus on lightweight, runtime detection frameworks capable of countering diverse evasion techniques in real-world scenarios. Another essential direction involves breaking steganographic concealment directly on the device and developing countermeasures that mitigate reputation damage to repackaged goodware.

To improve the research landscape, adopting best practices for studies on VirusTotal — including time-bounded experiments and optimized loaders — will enhance reproducibility and validity. Strengthening collaboration between researchers and the cybersecurity industry is essential to tackle the growing sophistication of Android malware and secure the mobile ecosystem effectively.

6. Acknowledgments

This work was partially funded by the project I-Nest (G.A. 101083398 - CUP B97H22004950001) - Italian National hub Enabling and enhancing networked applications and Services for digitally Transforming Small, Medium Enterprises and Public Administration.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] Kaspersky, Attacks on mobile devices significantly increase in 2023, https://www.kaspersky.com/about/press-releases/attacks-on-mobile-devices-significantly-increase-in-2023, 2023.

[2] SpyCloud, The rise of mobile malware: How it's evolving and what to watch for, 2023. URL: https://spycloud.com/blog/rise-of-mobile-malware/, accessed: 2024-11-28.

[3] Google, Enhanced google play protect: Real-time protections for your android apps, https://security.googleblog.com/2023/10/enhanced-google-play-protect-real-time.html, 2023.

[4] Google, New real-time protections on android, https://security.googleblog.com/2024/11/new-real-time-protections-on-Android.html, 2024.

[5] U. Kargén, N. Mauthe, N. Shahmehri, Characterizing the use of code obfuscation in malicious and benign android apps, in: Proceedings of the 18th International Conference on Availability, Reliability and Security, 2023, pp. 1–12.

[6] S. Aonzo, G. C. Georgiu, L. Verderame, A. Merlo, Obfuscapk: An open-source black-box obfuscation tool for android apps, SoftwareX 11 (2020) 1–6. doi:10.1016/j.softx.2020.100403.

[7] Guardsquare, ProGuard, https://www.guardsquare.com/proguard, 2024.

[8] Beijing Zhiyou Network Security Technology Co., Ltd, Ijiami, http://www.ijiami.cn/, Accessed in 2024.

[9] Bangcle Security, Bangcle, http://www.bangcle.com/, Accessed in 2024.

[10] Baidu Security, Baidu APK Protect, http://apkprotect.baidu.com/, Accessed in 2024.

[11] D. Dell'Orco, G. Bernardinetti, G. Bianchi, A. Merlo, A. Pellegrini, Would you mind hiding my malware? building malicious android apps with stegopack, SSRN Pre-Print (2024). doi:10.2139/ssrn.5039499.

[12] L. Caviglione, W. Mazurczyk, Never mind the malware, here's the stegomalware, IEEE Security & Privacy 20 (2022) 101–106. doi:10.1109/MSEC.2022.3178205.

[13] Dr.Web, Necro android malware found in popular apps, https://news.drweb.com/show/?lng=en&i=11685&c=5, 2024.

[14] G. Cluley, Android trojan using steganography for concealment, https://grahamcluley.com/android-trojan-steganography/, 2024.

[15] T. H. News, Necro android malware found in popular apps, https://thehackernews.com/2024/09/necro-android-malware-found-in-popular.html, 2024.

[16] G. Bernardinetti, D. Di Cristofaro, G. Bianchi, PEzoNG: Advanced packer for automated evasion on windows, Journal of computer virology and hacking techniques 18 (2022) 315–331. doi:10.1007/s11416-022-00417-2.

[17] R. Winsniewski, Android–apktool: A tool for reverse engineering android apk files, 2012. URL: https://apktool.org/.

[18] EvilSocket, Android applications reversing 101, https://www.evilsocket.net/2017/04/27/Android-Applications-Reversing-101/, 2017.

[19] M. Zheng, P. P. C. Lee, J. C. S. Lui, Adam: An automatic and extensible platform to stress test android anti-virus systems, in: U. Flegel, E. Markatos, W. Robertson (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 82–101.

[20] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, F. Mercaldo, Detection of obfuscation techniques in android applications, in: Proceedings of the 13th International Conference on Availability, Reliability and Security, 2018, pp. 1–9.

[21] M. Conti, P. Vinod, A. Vitella, Obfuscation detection in android applications using deep learning, Journal of Information Security and Applications 70 (2022) 103311.

[22] Y. Wang, H. Wu, H. Zhang, A. Rountev, Orlis: Obfuscation-resilient library detection for android, in: Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, 2018, pp. 13–23.

[23] J. Garcia, M. Hammad, S. Malek, Lightweight, obfuscation-resilient detection and family identification of android malware, ACM Transactions on Software Engineering and Methodology (TOSEM)

26 (2018) 1–29.
[24] RedNaga, APKiD, https://github.com/rednaga/APKiD, ongoing.
[25] A. Salem, F. F. Paulus, A. Pretschner, Repackman: A tool for automatic repackaging of android apps, 2018.
[26] K. Khanmohammadi, N. Ebrahimi, A. Hamou-Lhadj, R. Khoury, Empirical study of android repackaged applications, Empirical Software Engineering 24 (2019) 3587–3629.
[27] Y. Zhou, X. Jiang, Dissecting android malware: Characterization and evolution, in: 2012 IEEE symposium on security and privacy, IEEE, 2012, pp. 95–109.
[28] S. Rastogi, K. Bhushan, B. Gupta, Measuring android app repackaging prevalence based on the permissions of app, Procedia Technology 24 (2016) 1436–1444.
[29] Y. Ishii, T. Watanabe, M. Akiyama, T. Mori, Appraiser: A large scale analysis of android clone apps, IEICE TRANSACTIONS on Information and Systems 100 (2017) 1703–1713.
[30] Y. Zhang, X. Luo, H. Yin, Dexhunter: toward extracting hidden code from packed android applications, in: Computer Security–ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II 20, Springer, 2015, pp. 293–311.
[31] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, D. Gu, Appspear: Bytecode decrypting and dex reassembling for packed android malware, in: Proceedings of the 2015 International Symposium on Recent Advances in Intrusion Detection, Springer, 2015, pp. 359–381.
[32] LaurieWired, Badunboxing, https://github.com/LaurieWired/BadUnboxing, 2024.
[33] V. Sihag, M. Vardhan, P. Singh, A survey of android application and malware hardening, Computer Science Review 39 (2021) 100365.
[34] T. Muralidharan, A. Cohen, N. Gerson, N. Nissim, File packing from the malware perspective: Techniques, analysis approaches, and directions for enhancements, ACM Computing Surveys 55 (2022) 1–45.
[35] T. Morkel, J. H. Eloff, M. S. Olivier, An overview of image steganography., in: ISSA, volume 1, 2005, pp. 1–11.
[36] S. Badhani, S. Muttoo, Evading android anti-malware by hiding malicious application inside images, International Journal of System Assurance Engineering and Management 9 (2017). doi:10.1007/s13198-017-0692-7.
[37] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, Stegomalware: Playing hide and seek with malicious components in smartphone apps, in: D. Lin, M. Yung, J. Zhou (Eds.), Information Security and Cryptology, Springer International Publishing, Cham, 2015, pp. 496–515.
[38] F. Pagano, L. Verderame, A. Merlo, Obfuscating code vulnerabilities against static analysis in android apps, in: IFIP International Conference on ICT Systems Security and Privacy Protection, Springer, 2024, pp. 381–395.
[39] Android Developers, InMemoryDexClassLoader, https://developer.android.com/reference/dalvik/system/InMemoryDexClassLoader, 2024. [Accessed: 2024-08-30].
[40] S. Gupta, A. Goyal, B. Bhushan, Information hiding using least significant bit steganography and cryptography, International Journal of Modern Education and Computer Science 4 (2012) 27.
[41] VirusTotal, Virustotal: Free online virus, malware and url scanner, https://www.virustotal.com/, Accessed in 2024.
[42] S. Aonzo, Obfuscapk - libencryption plugin. apkid pull request #203, https://github.com/rednaga/APKiD/pull/203, 2020.