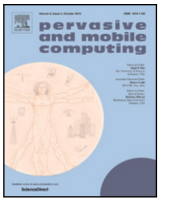
Pervasive and Mobile Computing 111 (2025) 102060

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc



Would you mind hiding my malware? Building malicious Android apps with StegoPack

Danilo Dell'Orco [a], Giorgio Bernardinetti [a], Giuseppe Bianchi [b], Alessio Merlo [c,*], Alessandro Pellegrini [b]

[a] CNIT - National Inter-University Consortium for Telecommunications, Via del Politecnico, 1, Rome, 0133, Lazio, Italy
[b] University of Rome Tor Vergata, Via del Politecnico, 1, Rome, 00133, Lazio, Italy
[c] CASD - School of Advanced Defense Studies, Piazza della Rovere, 83, Rome, I-00165, Lazio, Italy

ARTICLE INFO

Keywords:
Mobile security
Android stegomalware
Packing

ABSTRACT

This paper empirically explores the resilience of the current Android ecosystem against stegomalware, which involves both Java/Kotlin and native code. To this aim, we rely on a methodology that goes beyond traditional approaches by hiding malicious Java code and extending it to encoding and dynamically loading native libraries at runtime. By merging app resources, steganography, and repackaging, the methodology seamlessly embeds malware samples into the assets of a host app, making detection significantly more challenging. We implemented the methodology in a tool, StegoPack, which allows the extraction and execution of the payload at runtime through reverse steganography. We used StegoPack to embed well-known DEX and native malware samples over 14 years into real Android host apps. We then challenged top-notch antivirus engines, which previously had high detection rates on the original malware, to detect the embedded samples. Our results reveal a significant reduction in the number of detections (up to zero in most cases), indicating that current detection techniques, while thorough in analyzing app code, largely disregard app assets, leading us to believe that steganographic adversaries are not even included in the adversary models of most deployed defensive analysis systems. Thus, we propose potential countermeasures for StegoPack to detect steganographic data in the app assets and the dynamic loader used to execute malware.

1. Introduction

Mobile malware detection systems and antivirus (AV) engines rely predominantly on static analysis techniques [1], which involve signature-based detection methods that compare parts of an app against a database of known malware signatures. From an attacker's perspective, concealing malware during injection and loading processes is crucial to evading AV detection and bypassing security controls.

Steganography [2], the practice of embedding hidden information within ordinary data such as images or audio, naturally emerges as a powerful complement to traditional AV evasion methods such as code obfuscation or payload encryption [3–11]. When applied to malware, this technique is often called stegomalware [12]. Stegomalware hides the malicious payload within app components

* Corresponding author.
E-mail addresses: danilo.dellorco@cnit.it (D. Dell'Orco), giorgio.bernardinetti@cnit.it (G. Bernardinetti), giuseppe.bianchi@uniroma2.it (G. Bianchi), alessio.merlo@unicasd.it (A. Merlo), alessandro.pellegrini@uniroma2.it (A. Pellegrini).
URLs: http://netgroup.uniroma2.it/GiuseppeBianchi/biografia.html (G. Bianchi), https://www.csec.it/people/alessio_merlo (A. Merlo), https://alessandropellegrini.it/ (A. Pellegrini).

https://doi.org/10.1016/j.pmcj.2025.102060
Received 28 November 2024; Received in revised form 11 March 2025; Accepted 19 April 2025
Available online 9 May 2025

typically ignored by static analysis, such as non-code resources. The malicious code is extracted and executed only at runtime, rendering it invisible to static analysis tools. Unlike encrypted malware loaders [13], stegomalware does not necessarily introduce additional data or exhibit unusually high entropy, common indicators used by "standard" static analysis for detection.

Despite extensive research conducted over the decades on steganographic algorithms and detection techniques (see, e.g., [14–18], among many earlier works), its specific impact on the Android ecosystem has received relatively limited attention [19,20]. We posit that one possible reason for this limited interest may be that ultimately, any stegomalware—regardless of the specific steganographic technique used—must eventually rely on some method to *load* the malware. Consequently, defensive techniques can potentially disregard steganography and concentrate their detection efforts *only* on this loading phase.

With traditional DEX-based malware samples, the above defensive strategy (targeting "just" the loading phase) appears straightforward and may be reasonably reliable. The vast majority of packers designed for traditional APK apps [21–24] rely, at some point, on dynamic loading of the malicious or repacked code via the `InMemoryDexClassLoader`. Hence, its invocation can be considered as an indicator of compromission[1]—indeed, as shown later on in Section 5.7, this strategy, duly adapted, is currently used by the well-known ESET Antivirus.

However, we argue that the ever-increasing deployment of *native* apps—those that rely, either partially or entirely, on shared binary libraries or architecture-specific native code—dramatically alters the situation described above. This shift is driven by the availability of a significantly larger arsenal of loading techniques, as packer developers are no longer restricted to methods necessarily compatible with `dex` files.

Motivated by the above, with this paper, we aim to address three complementary research questions:

- **RQ1**: how viable is stegomalware within the evolving Android apps deployment ecosystem, thus including (also) native apps?
- **RQ2**: how prepared is today's Android defense ecosystem, e.g., commercial AVs and Google Play Protect, to counter a potential widespread stegomalware adoption?
- **RQ3**: which remediation strategies should AV engines adopt to address the rise of stegomalware better?

We address **RQ1** by (i) experimentally evaluating the capacity of Android apps in the wild to embed sufficient assets to reliably hide malicious payloads (Section 5.4) and (ii) developing a proof-of-concept steganographic malware loader (Section 4). Our tool, named StegoPack, integrates steganography with app repackaging, distributing the malicious payload across the app's assets (e.g., non-compiled resources) and employing a resource merging and loading strategy to install the malware at runtime. Notably, to our knowledge, StegoPack is the first demonstration of a stegomalware instance (also) capable of operating on native apps.

Using StegoPack, we address **RQ2** by conducting extensive real-world testing (Section 5). We embedded various malware samples (including `dex` files and native shared objects), spanning over 14 years, into popular and lesser-known host apps. These samples, with high detection rates on VirusTotal [29], showed a detection rate drop to nearly zero after being embedded using StegoPack. Via tailored and repeated experiments, we provide evidence that either the 79 tested antivirus engines as well as Google's Playstore (a packed app passing Play Console security checks, see Section 5.9) fail even to consider the possibility of malware being steganographically embedded into app assets—in other words, steganographic adversaries appear not to be in their adversary model.

Finally, we address **RQ3** in Section 6. Our main conclusion is that relying solely on steganalysis is insufficient to detect Android stegomalware due to a high rate of false positives, particularly from small or patterned images that characterize several apps (Section 6.2). Hence, the *combination* of different dynamic analysis approaches to recognize the steganographic loader that builds and executes the malicious payload at runtime, with a *further* steganalysis stage, appears to be a more compelling strategy for countering stegomalware.

2. Background

Android basics. Android's architecture (Fig. 1) consists of a Java stack on top of a general-purpose Linux kernel, which manages system resources like memory, processes, and device drivers. Above the kernel, the Hardware Abstraction Layer (HAL) provides a standard interface for higher-level Java APIs to interact with physical resources.

The Android RunTime (ART) provides the application execution environment, running Dalvik Executable format (`dex`) files compiled from Java/Kotlin source code. ART uses Ahead-of-Time Compilation (AOT) at installation to convert `.dex` files into native machine code (`.art` and `.vdex`), stored in the `/data/dalvik-cache` directory.

Each app runs with a unique user ID (UID) to ensure isolation. Android also includes native C/C++ libraries that provide optimized services (2D/3D graphics, audio/video codecs, etc.). The Java API Framework, a standard non-privileged UID, supports app execution. At runtime, methods invoked in the Android SDK trigger corresponding services in the Java API Framework, such as the Activity and Package managers for app lifecycle, installation, permission management, and the Resource manager for asset access.

Architecture of an app. The Kotlin/Java code of an Android app is 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

[1] Though detection of `dex` files loading into memory via `InMemoryDexClassLoader` obviously cannot be used as the *sole* IoC, since this technique is employed in several *defensive* techniques such as code obfuscation methodologies designed to counteract reverse engineering and protect the original code from manipulation [25–28].

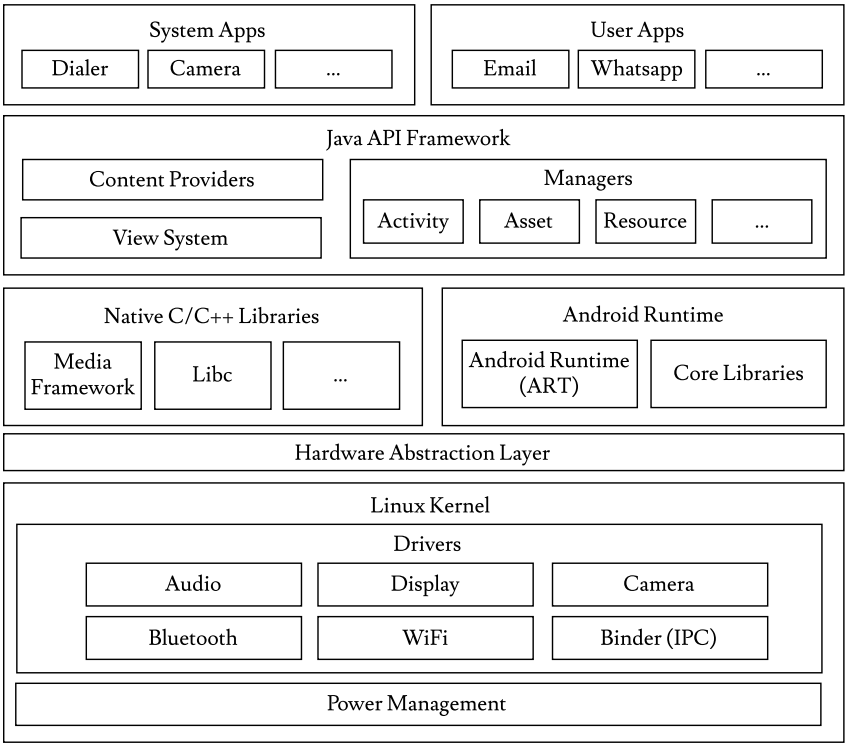


Fig. 1. The Android software stack.

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 (assets).

Fig. 2 shows the architecture of an APK [30]. The main elements of an APK are *dex files*, *resources*, *shared libraries*, *assets*, and the *AndroidManifest.xml* 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.

The `res/` folder is organized into several subdirectories: `layout/`, which contains XML files defining the structure of the app's UIs; `drawable/`, which holds binary images and shapes used in the UI; and `raw/`, which includes raw data files like image, audio, or configuration files. A unique numerical ID identifies each resource managed in the `public.xml` file.

Shared libraries, which are stored in the `lib/` folder, consist of ELF binaries (*.so* files) built by compiling architecture-specific native code (C/C++).

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.

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.

Modifying Android Apps. APK files can be reversed using decompiling tools (e.g., ApkTool [31]) 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. ApkTool also generates an `apktool.yml` file with configuration settings for recompiling the app.

3. Related work

Repackaging. Repackaging involves reverse engineering an APK, adding code or resources, recompiling, resigning, and redistributing it in the wild. Attackers use repackaging to insert malicious payloads into popular Android apps and redistribute them through official marketplaces or side channels (e.g., websites or email), tricking users into installing the altered version [32–34]. To counteract repackaging, developers may use anti-repackaging techniques that cause the app to crash if anti-tampering checks on the original code fail. Merlo et al. [35] analyzed these techniques, demonstrating how to bypass all checks performed by NRP, a publicly available anti-repackaging tool. They also designed ARMAND [36], a new anti-repackaging scheme utilizing multiple

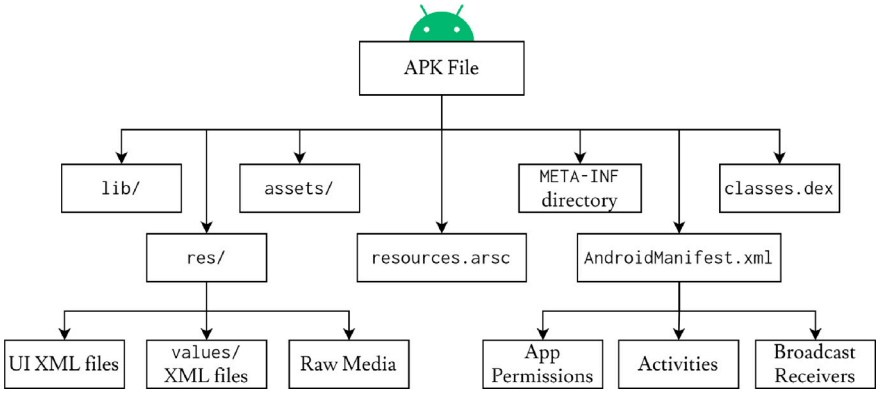


Fig. 2. Structure of a compiled Android APK.

protection patterns and native code to overcome limitations of existing methods. The approach showed robustness against common attack vectors, at the price of occasional (3.2%) app crashes.

Repackman [21] is a tool that automatically repackages Android apps with arbitrary payloads, creating fake, malicious versions of legitimate apps to test anti-repackaging techniques. Experimental results showed that *Repackman* successfully repackaged 86% of Google Play apps with no noticeable side effects. Commercial packers such as Baidu [22], Bangcle [23], and Ijiami [24] allow reverse engineering tools to hinder app analysis. They can further thwart malware analysis by encrypting the original app's `classes.dex`, packing it into the APK, and decrypting it at runtime using `DexClassLoader`. However, obfuscation and packers can be countered by tools such as DexHunter [37] and AppSpear [38], which can extract original bytes from *dex* files packed by established packers, or tools like APKiD [39], which use specific Yara rules to automatically detect well-known obfuscators and packers. Moreover, using tools like *Repackman* to add malware and repackage apps leaves detectable non-obfuscated code in the host's smali.

Obfuscation. Obfuscation [5] refers to techniques that make code difficult for reverse engineers to understand. In Android, benign apps (to protect intellectual property) and malware apps (to hide malicious payloads) commonly use obfuscation due to the ease of decompiling apps. Zheng et al. conducted a study showing that automated malware obfuscation initially decreased antivirus detection rates, but after four months, detection rates surged from 54% to 90%, indicating growing antivirus adaptation [40]. Indeed, commercial antivirus products have evolved to detect obfuscation better. The QuickHeal Malware report of 2020 [41] highlighted that the `Android.Obfus.GEN36238` threat, explicitly designed to target obfuscated apps, accounted for 8% of Android malware. Various solutions now label different obfuscated apps, such as `Android/Obfus` [42–44].

The research community has also made extensive efforts to detect obfuscated malware. Various studies [45–48] propose static analysis methods for identifying obfuscation techniques. Additionally, DroidPDF [49], using an entropy-based approach, is capable of detecting repackaged and obfuscated apps, tested against Proguard and ObfuscAPK. Research work by Graux et al. [50] and Kargen et al. [5] has examined the effectiveness of various obfuscation techniques, noting that some methods, like string encryption, tend to produce high false positive rates by misidentifying benign obfuscated apps as malicious.

Steganography. Steganography refers to techniques used to covertly embed information in communication channels or files, such as images or audio. A standard method is the Least Significant Bit (LSB) scheme [51], which alters the least significant bit of pixel values to encode hidden data. This technique exploits the imperceptibility of small changes in the color channels of the pixels. In addition to being used for legitimate purposes, such as discreet data transmission, steganography can develop stegomalware for code concealment, payload delivery, and establishing covert command and control channels. Several studies [14–18,52,53] proposed solutions (often based on probabilistic or machine learning methods, due to the difficulty in determining the attacker's algorithm) to detect benign image manipulation using LSB [54].

Regarding Android, several studies highlighted the potential abuse of steganography for malicious purposes. For instance, Spreitzenbarth et al. [55] analyzed the `FakeRegSMS` malware, which embeds encrypted malicious code within the app icon. The study suggested a shift towards a new era of malware obfuscation, with steganography being a valuable alternative to traditional code manipulation methods. Badhani et al. [20] demonstrated the efficacy of steganography in evading Android antivirus by showing that hiding a malicious app within images evaded detection by nine out of ten antivirus tools. On the defensive side, Suárez-Tangil et al. [19] proposed a detection method combining asset steganalysis with dynamic code loading and steganographic decoding analysis. Their static analysis-based mechanism estimates the likelihood of an app being stegomalware. Their findings highlight the difficulty of detecting malicious information extraction, as many legitimate apps manipulate images similarly to malicious LSB decoding algorithms. Concerning remediation, limited research exists on the behavioral analysis of Android stegomalware, particularly its implementation. The detection method in [19] targets only specific steganography algorithms and relies solely on static analysis, which struggles to distinguish steganographic malware from legitimate image manipulation software.

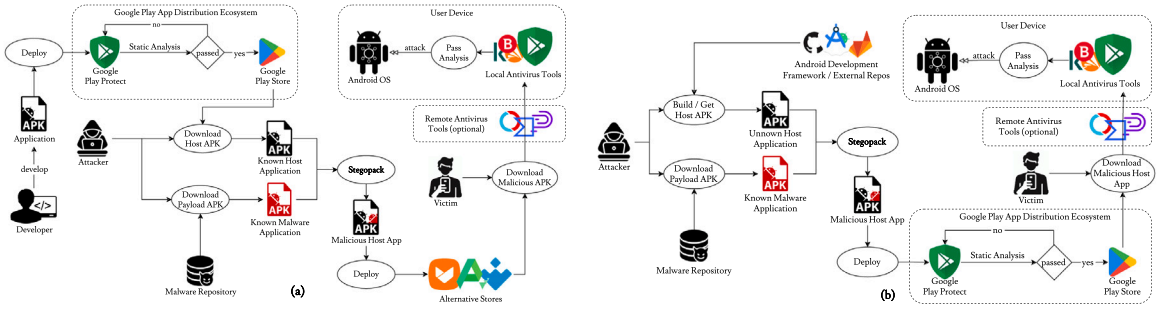


Fig. 3. StegoPack: Threat model using (a) a known app and (b) an unknown app as host.

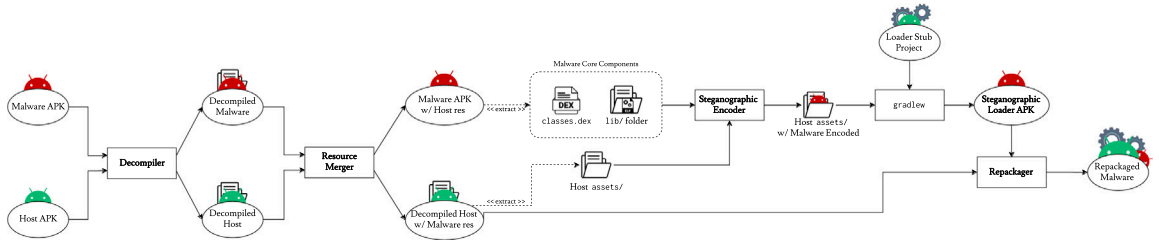


Fig. 4. StegoPack: Malware repackaging workflow.

4. Approach and methodology

4.1. Threat model

Our threat model considers an attacker embedding a malware sample into a host app, aiming for the victim to install it on their device. We distinguish two scenarios based on the host app: either a *known* app (already published on the Google Play Store) or an *unknown* app (new or unpublished on the store).

Fig. 3(a) refers to the first case: here, a developer implements and distributes a benign app on the Google Play Store. The app passes all checks (e.g., the Google Play Protect analysis) and is published. The attacker downloads the app and an existing malware sample from some external repositories and applies StegoPack to create the malicious host app by embedding the malware into the host app. Then, the attacker redistributes it through some alternative sources (e.g., alternative app stores or via the web). The victim downloads and installs it on her device. Before installation, the victim may submit the APK to some external AV engines. During installation and execution, other local AVs (e.g., Google Play Protect) may check the APK. The attack is successful if the malicious host app bypasses all checks and executes correctly on the victim's device.

Fig. 3(b) refers to the case of unknown host apps. In this case, the attacker builds or retrieves an unpublished app from some external sources and the malware sample, thereby building a new malicious host app through StegoPack. Then, she uploads the app to the Google Play Store. Here, the attacker wins if the malicious host app also bypasses the checks on the Google Play Store, and the malicious host app is eventually published as a benign app.

4.2. Stego packaging workflow

As illustrated in Fig. 4, we envision a workflow comprising five main components, each corresponding to a conceptual stage of the malware embedding process.

Decompiler. The process starts by decompiling the host and malware APKs using an off-the-shelf *Decompiler* (i.e., ApkTool) to extract the associated smali code and unencrypted resources, as described in Section 2. This step enables the manipulation of the different APK resources.

Resource Merger. This stage integrates resources from the host and payload APKs while resolving conflicts to ensure seamless integration. This step is crucial to maintain a coherent and functional APK. It guarantees that when executed within the host's

context, the payload can access all its original resources without error. The core payload components are extracted directly from the APK without decompiling. These include the `classes.dex`[2] file and the `lib/` folder containing native libraries.

Steganographic Encoder. This stage applies a steganographic algorithm to embed the payload files extracted in the previous merging stage in the host's image assets. Since the main focus of this paper is proving that elementary steganography suffices to hide malicious payload reliably, we relied on a baseline Least Significant Bit (LSB) steganography algorithm [56] (for a basic background please refer to Appendix A), with the additional capabilities of embedding payloads over multiple images, and supporting a variable embedding rate (of up to *2 bits per pixel - bpp*). We embed the payload's `classes.dex` file and all compiled native libraries (`.so` files), making them inaccessible to static analysis. Although embedding the `classes.dex` file is straightforward since it is a single file, for native libraries, StegoPack uses a special header to structure the encoding of the `lib/` folder, specifying library names, CPU architectures, and file sizes to ensure accurate reconstruction during decoding. Refer to Appendix B for further details.

Steganographic Loader. The crucial part of any stegomalware is the loader, whose task is to extract the payload files from the host's assets and execute them at runtime. As discussed in Section 4.5, the loading approach significantly differs between `dex` and native cases.

Repackager. This final component inserts the steganographic loader, containing the encoded payload, back into the host app. This step also involves embedding the necessary permissions for the payload into the host's manifest 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. See Section 4.4 for further implementation details.

4.3. Resource merger

The Resource Merger (Fig. 5) is responsible for integrating the malware's resources (such as `layouts`, `drawables`, and `raw` files) into the host application while preventing conflicts between resource names and identifiers. A simple copy operation would result in naming conflicts if the host and malware share files with identical names and identifier conflicts within Android's resource management system.

Conflicts are managed at two levels to prevent these issues: file names and resource identifiers. Naming conflicts may occur both at the file level, where different resources share the same filename, and within XML files, where resource entries might have identical names. To resolve file-level conflicts, the merger appends the suffix "`_c`" to each copied malware resource, ensuring it does not overwrite existing files. For XML-based resources within the `res/values/` directory, each element must be merged into a single unified file, as Android assigns a unique scope to these entries (see Section 2). If an XML element is missing from the host's file, it is added directly. However, if a name conflict arises within the XML structure, the payload attribute's name is modified by appending "`_c`" before being integrated.

Beyond naming conflicts, Android resource IDs must also remain unique. Both the host and malware contain a `public.xml` file, which assigns fixed resource IDs in the format `0xPPTTEEEE`, where `PP` denotes the package, `TT` the resource type, and `EEEE` the unique identifier. Since the `TT` type codes are not standardized across different apps, identical resource types may have different values, and identical `EEEE` values could lead to conflicts. To ensure consistency, StegoPack first aligns the `TT` bytes between host and payload resources. If conflicts persist due to overlapping `EEEE` values, StegoPack increments the conflicting payload resource ID incrementally until a unique value is assigned.

Once conflicts are resolved, the necessary modifications must be propagated throughout the app to ensure consistency. In Android, resource references span multiple files, requiring systematic updates. StegoPack tracks all changes introduced during the merging process and applies them collectively in a dedicated phase. Resource name updates are propagated across all XML files using a regex-based search-and-replace strategy, including `AndroidManifest.xml`. Additionally, since resources are accessed by their identifiers in the smali code, adjustments made during `public.xml` resolution are reflected in the payload's decompiled smali files to maintain correct resource loading during execution.

4.4. Repackaging

StegoPack uses a repackaging attack to integrate malware functionality into the host app. Previous methods, such as those described in [21], embed the malicious code directly into the host app. This approach is effective only for simple malware with limited code modifications and is impractical for complex malware with multiple activities. The repackaging approach in StegoPack avoids direct embedding of malware. Instead, StegoPack injects only the steganographic loader app.

This approach significantly reduces the modifications of the host app by incorporating just a single additional activity. Since the payload is steganographically encoded, there are no discernible traces of the malware's APIs within the repackaged host code. StegoPack leverages specific user interactions to execute the malicious payload within the host app by modifying an existing host activity. To this aim, StegoPack modifies the host's *main activity* by customizing the implementation of the `onCreate()` method in the corresponding `smali` file. Here, StegoPack inserts an `invoke-static` call to activate the steganographic loader, which decodes and executes malware.

[2] Note that at this stage, `classes.dex` is not the original file but has updated resource references reflecting the merging process.

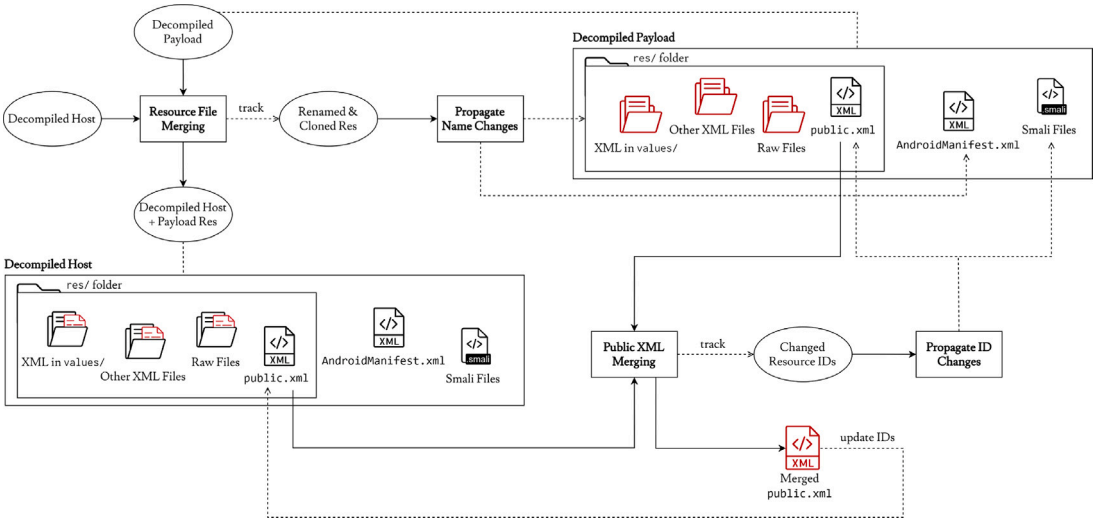


Fig. 5. StegoPack: Resource merger scheme.

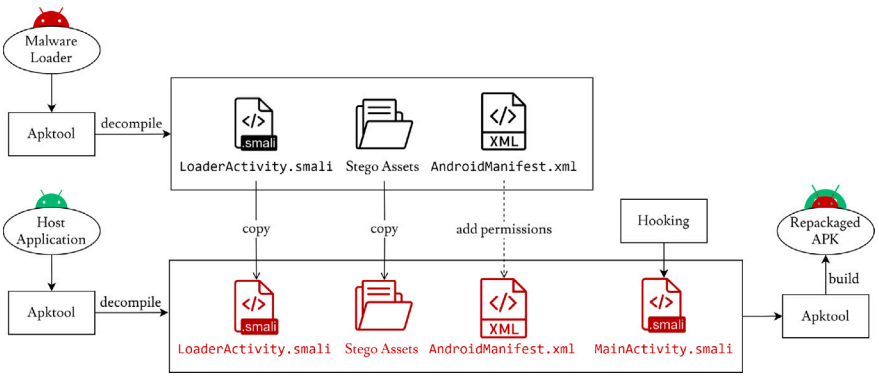


Fig. 6. StegoPack: Workflow of the repackaging process.

Upon execution from the loader, the payload works as a standard Activity within the host context. Consequently, the manifest file referenced at runtime is that of the original app, encompassing all permissions necessary for the host but lacking those specific to the payload. To ensure that the payload could be executed at runtime, it becomes imperative to verify the presence of essential permissions within the `AndroidManifest.xml` file of the host app, ensuring alignment with the payload's requirements.

Our approach involves mirroring all permissions specified in the payload manifest onto the host side. This process is initiated by parsing the payload's manifest, extracting the required permissions, and subsequently appending them to the host manifest's relevant `intent-filter` element. To maintain manifest integrity and prevent potential conflicts or duplicates, each payload permission is verified for existence in the host's manifest before inclusion (see Fig. 6).

4.5. Strategies for dynamic code loading

Existing stegomalware techniques [19,20] primarily focus on concealing Java code within an app's assets, while leaving native libraries exposed, allowing them to be easily identified through static analysis, thereby resulting in a significant portion of the malicious payload vulnerable to detection.

To overcome this limitation, our methodology (see Fig. 7) also *shields native components from static analysis*, thereby aligning stegomalware with modern threats that leverage native code for malicious actions. Although loading Java code into the environment requires specific APIs (such as the `DexClassLoader` family), attackers have greater flexibility in loading and executing native code, which can be achieved through various methods. As described in the following, our methodology relies on established techniques for Java code loading while introducing novel methods for handling native components.

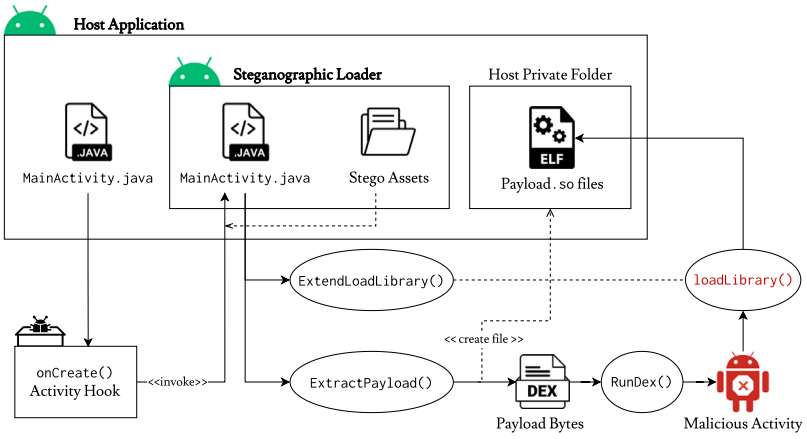


Fig. 7. StegoPack: Payload extraction and loading process.

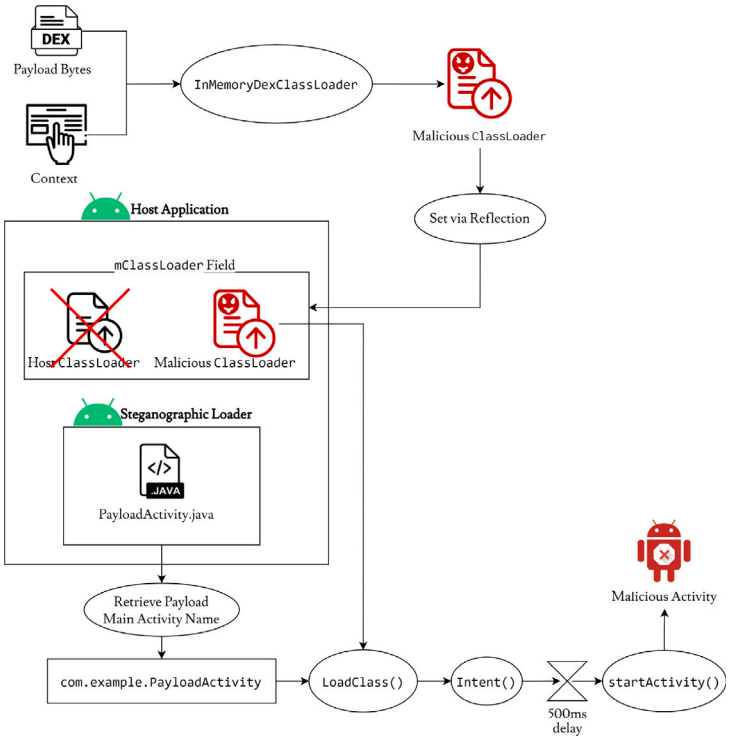


Fig. 8. Substitution of the ClassLoader and Runtime Dispatching of the payload activity from the host.

4.5.1. Java code loading

Concerning Java, our approach aligns with the current state of the art, which typically involves using class loaders from the `DexClassLoader` family. Such APIs allow the integration of dynamically loaded code into the runtime environment of the host app without leaving detectable traces for static analysis. Specifically, we use the `InMemoryDexClassLoader` for dynamically loading `dex` files that were not initially part of the APK. During the decoding phase, the loader reverses the steganographic encoding algorithm to extract the bytes of the `classes.dex` file from the host's assets.

The next responsibility of the loader is to start the payload activity at runtime. This involves several steps, depicted in Fig. 8. First, it implements a custom `ClassLoader` by creating an `InMemoryDexClassLoader` object with the payload's `classes.dex` bytes and the parent `ClassLoader`, enabling delegation to load both malicious and host app classes. Java Reflection is extensively exploited to access `mClassLoader` within the host APK class, and the custom `ClassLoader` with payload classes is assigned to this field. Next, using the malicious `ClassLoader`, the host app dynamically loads the class `com.example.package.MaliciousActivity` through the `loadClass()` method. The string to specify the payload activity

is not passed statically as a parameter. Still, it is dynamically obtained at runtime from a dedicated `PayloadActivity.java` file that masks the malware activity name as image filenames in a Java array to avoid static detection. After deriving all the original bytes, the loader's next task is to start the payload activity at runtime. To evade detection by algorithms monitoring activity transitions, a deliberate 500 ms delay is introduced. This is done by instantiating a handler that targets the main looper, enabling delayed execution by scheduling a runnable task. In the current context, the task activates the malicious payload's activity using the `startActivity()` method. This intentional delay disrupts expected timing, reducing detection rates and enhancing evasion capabilities.

4.5.2. Native library loading

Our approach extends traditional methods by managing native libraries, which are increasingly adopted in modern Android malware [57]. Unlike Java code, which benefits from robust support for dynamic loading through existing Android APIs, dynamic loading of native libraries, especially those extracted from steganographically hidden resources, remains largely unexplored in stegomalware.

The core strategy involves making malicious ELF files available only at runtime, evading static analysis. This is achieved by having the Steganographic Loader extract the bytes of each native library and write them to corresponding `.so` files within the host's private directory under `/data/data`. Storing the libraries in this location offers *two key advantages*: it restricts access, ensuring that other apps cannot detect or access the files, and it enables the host to load libraries that were not originally packaged in the APK, all without requiring explicit user permissions.

To implement this strategy, we modify the conventional process of loading native libraries in malware apps. Normally, when an Android app invokes the `System.loadLibrary()` function, it searches for a file named `lib<Name>.so` in predefined directories [58], primarily within the APK's private directory at `/data/app/<hash1>/<hash2>/lib`. In our pipeline, however, the payload's `.so` library is deliberately excluded from the repackaged APK to conceal its presence. Since the original APK directory has read-only permissions, it cannot be used to store or extract the payload. Consequently, any attempt to load the library using `System.loadLibrary()` fails due to the missing `.so` file.

Instead, we extract the original library files from stego-images, writing the `.so` files into the `/data/data` directory and subsequently loading them from there. Based on the library name provided in the original `loadLibrary()` call, we determine the CPU architecture of the device and reconstruct the full path as follows:

`/data/data/com.host.app/files/lib/<cpu_arch>/lib<name>.so`.

We propose two possible strategies to enable the original application to load libraries from this custom path, both achievable **without requiring root permissions**. The first involves modifying the Android Runtime to support the new path, while the second extends the ClassLoader to allow library loading from additional directories beyond the default ones.

Hacking Android Runtime. This approach is based on replacing the standard `System.loadLibrary()` method with a custom implementation that leverages `System.load()` [59]. Unlike `System.loadLibrary()`, which requires only the library name, `System.load()` allows loading a `.so` file by specifying its full path. Internally, this API calls the native `dlopen` function, which restricts library loading to directories specified in `permitted_paths` [60]. One such permitted path is the app's private directory (`/data/data/com.host.app`), where the host app has write permissions.

To achieve this, every invocation of `System.loadLibrary()` must be redirected to the custom implementation. Instead of modifying each call individually, we manipulate Android Runtime (ART) to alter the execution behavior of the original `System.loadLibrary()` method. Following the approach proposed by ARTFul [61,62], this is done by replacing the runtime representation of the `System.loadLibrary()` method, stored as an `artMethod` object in the `Executable.java` file. Using the `memcpy` function via JNI, the memory region containing the `artMethod` structure for `loadLibrary` is overwritten with the corresponding structure for the custom implementation, effectively altering its behavior at runtime.

The entire process is implemented within a custom library named `LLHook`, compiled and embedded into the host app during the repackaging phase. Upon first startup of the host app, `LLHook` is loaded using the original `System.loadLibrary()` method. Once loaded, `LLHook` replaces the default `System.loadLibrary()` with the custom version. From that point on, all calls to `System.loadLibrary()` are seamlessly redirected to `System.load()`, enabling the loading of the reconstructed `.so` file extracted during the decoding phase. The workflow of this mechanism is illustrated in Fig. 9.

A key advantage of this approach is its ability to directly modify the Android Runtime, allowing for the redirection of other APIs in addition to `loadLibrary`. However, a notable limitation is the mandatory inclusion of `LLHook.so` within the repackaged app, which could serve as an indicator of compromise (IoC) for antivirus solutions.

Extending InMemoryDexClassLoader. The `InMemoryDexClassLoader` is restricted to searching for native libraries in directories specified by the `nativeLibraryDirectories` variable in the `java/dalvik/system/DexPathList.java` class. By default, this variable includes only system directories such as `/system/lib64` and `/system_ext/lib64`. Consequently, libraries extracted to private directories, such as `/data/data`, cannot be loaded using `System.loadLibrary()`.

However, starting from API Level 29, `InMemoryDexClassLoader` introduces an additional constructor that accepts a `librarySearchPath` parameter. This parameter specifies the directories from which the DEX file (in this case, the malicious payload) can search for native libraries. By setting this parameter to the host app's private directory (`/data/data/com.host.app`), the payload gains the ability to load its native libraries extracted at runtime using its original `System.loadLibrary()` invocations.

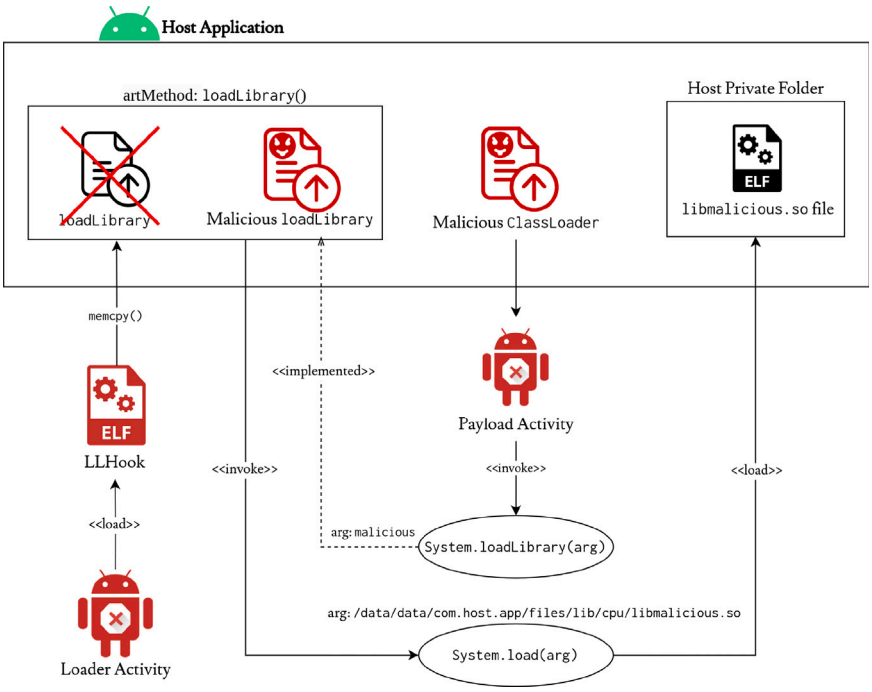


Fig. 9. StegoPack: Dynamic loading of native libraries.

Compared to previous approaches, this method offers a significant advantage: it eliminates the need to modify the payload application while ensuring no `.so` files are exposed in the repackaged app. The only adjustment required is including the private directory as a parameter when initializing `InMemoryDexClassLoader`. This subtle change makes the approach inherently stealthier from an attacker's point of view, as the payload loading behavior remains virtually identical to standard Java loading, aside from the additional parameter passed during DEX loader initialization.

5. Experimental evaluation

5.1. Datasets

For our analysis, we utilized three distinct datasets: the *Goodware Dataset*, the *Malware Dataset*, and the *Stegomalware Dataset*. The *Goodware Dataset* and *Malware Dataset* were used to gather statistical data regarding asset sizes, steganalysis, and embedding capacities. These datasets contain legitimate (goodware) apps and malware samples, respectively.

To evaluate the effectiveness of antivirus solutions against stegomalware, we constructed the *Stegomalware Dataset*. This dataset was generated by selecting a subset of malware and goodware host apps, which were repackaged in all possible combinations using StegoPack. The resulting dataset allows us to assess antivirus performance in steganographic payload embedding and examine how various antivirus tools detect malware when disguised within benign apps.

Malware Dataset. Consists of 13,195 actual Android malware samples, which include 2316 native apps and 10,879 Java-only apps, collected between 2011 and 2024. These samples were sourced from three major repositories: i) *VX Underground*, a dataset featuring malware samples from 2011 to 2017, ii) *github.com/sk3ptre*, containing malware specimens from 2018 to 2022, and (iii) *Malware Bazaar*, primarily including very recent (2022+) malware instances.

Goodware Dataset. The *Goodware Dataset* includes 274 benign apps collected from various sources, such as the Google Play Store, alternative app stores, public GitHub repositories, and in-house developed apps hosted on private GitHub repositories (never released). These apps were carefully selected to ensure they do not trigger any malware detections. All apps in this dataset were verified to produce *zero* detections when submitted to VirusTotal, ensuring their legitimacy as "goodware".

Stegomalware Dataset. To build our *Stegomalware Dataset*, we selected a set of host apps and malware payloads, repackaging each malware sample within each chosen host. This process resulted in a 14,000 stegomalware instances dataset generated using StegoPack.

The selection of malware payloads was based on the *malware dataset*, from which we extracted 140 distinct samples spanning 2011–2024. Specifically, we selected five native and five non-native malware samples per year, covering 119 unique malware families. To ensure the relevance of our dataset, each selected sample had at least 15 detections on VirusTotal. We deliberately

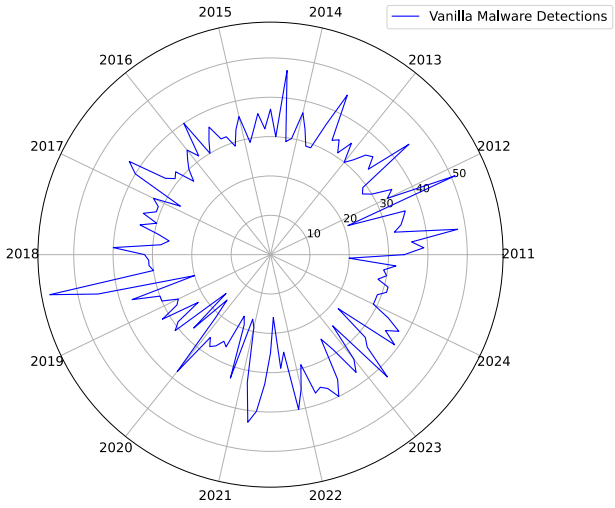


Fig. 10. Total detections for each original malware payload.

Table 1
Unmodified samples: Top 14 AV detection capability.

AV	# detection	perc.	AV	# detection	perc.
Trustlook	140	100.0%	Ikarus	133	95.0%
Avast Mobile	136	97.1%	McAfee	133	95.0%
Kaspersky	136	97.1%	Symantec	133	95.0%
K7GW	135	96.4%	Fortinet	132	94.3%
ESET-NOD32	135	96.4%	Dr. Web	128	91.4%
BitDefender Falx	135	96.4%	Avira	128	91.4%
Symantec Mobile Insight	134	95.7%	Lionic	127	90.7%

focused on well-known malware apps widely detected by antivirus (AV) engines, as our primary goal is to assess how AV solutions perform against StegoPack's steganography and repackaging technique. Similarly, to ensure diversity in host apps, we selected 100 legitimate APKs from the *goodware dataset*. These include 70 apps from the Google Play Store, 18 from public GitHub repositories, 8 from alternative app stores, and 4 unreleased apps. Our objective is to analyze how commercial antivirus software responds to modifications in both widely recognized and unfamiliar apps.[3]

Depending on the specific combination of apps, some selected host apps may lack sufficient assets to accommodate larger payloads. To ensure adequate embedding capacity, we inserted synthetic images into the `assets/icons` folder. We set the embedding rate for all these samples to a maximum of *2 bits per pixel*.

5.2. Vanilla malware assessment

To establish a baseline for comparing the original malware samples and their stegopacked counterparts, we analyzed the 140 selected payloads using VirusTotal. The radar plot in Fig. 10 shows the total detection rate for the sample. Out of 11,060 data points (140 malware tested against 79 antivirus tools), the analysis led to 4408 detections, resulting in a mean detection rate of about 39.85%. Table 1 illustrates the top 14 antivirus programs, ranked by their overall detection capabilities, which detected more than 90% of the 140 malware samples submitted. Remarkably, Trustlook performed flawlessly, detecting all submitted malware samples.

5.3. Packaging verification process

To evaluate the compatibility of various payload–host pairs using StegoPack, we implemented a pipeline with a two-step testing process:

1. **Repackaging Validation**: We repackage the payload–host pair using Stegopack, monitoring the process to ensure no errors. If errors occur, the pair is discarded. Successful repackaging leads to the next step.

[3] It is worth noting that using unknown apps aligns with our threat model, mirroring a scenario where a malicious actor attempts to distribute malware disguised as a legitimate new app.

2. **Functional Testing**: We execute the malware to verify that the repackaged host can correctly dispatch the payload at runtime. A testing device, either physical or an emulator, is initialized using the *Android Debug Bridge*. We then start `adb logcat` to monitor the app's behavior and launch the repackaged malware via a shell command. The app's runtime behavior is monitored for ten seconds, during which we look for potential crashes or errors. To confirm that the loader successfully launches the payload activity, we check for the presence of the message "`Displayed <payload_activity`" in the logcat, as recorded by the `ActivityTaskManager`.

Although all of the 14,000 submitted samples passed the *Repackaging Validation* phase without any errors, functional testing was limited to 280 samples selected from the Stegomalware dataset, which consists of two host apps, each paired with 140 malware samples. This reduction to 280 apps was necessary for practical reasons. Despite the automated nature of the testing pipeline, thoroughly testing each of the 14,000 sample pairs would require substantial computational resources, time, and manual effort. Testing involves multiple steps, including app installation, interaction stimulation, and monitoring of runtime behavior, all of which require considerable time per sample. Moreover, manual log analysis is critical for diagnosing and classifying crashes, as it helps differentiate between issues explicitly related to the stegopacking process and those caused by resource incompatibilities between the malware payload and the host app. Given the complexity of these tasks and the importance of capturing errors accurately, we chose a manageable subset of 280 instances for detailed testing. This approach ensures we can observe the malware's behavior comprehensively while maintaining an efficient testing process.

Failed execution of malicious host apps. Repackaged apps did not show crashes in 94 cases out of 280 (over 33%). Most errors occurred with recent malware (2017+) that crashed after being loaded by the host due to various issues: the app not running on the target device, incompatibility between the SDK compile versions of the host and payload, or missing resources and classes at runtime.

In detail, 45.71% of crashes involved `Resources$NotFoundException`, while 6.42% crashed during payload decoding due to JVM memory exhaustion. Additionally, 5.35% of cases encountered `ClassNotFoundException` when attempting to load one of the payload activities. In 7.5% of the cases, the payload, once loaded, failed to execute one of its functions, likely due to SDK version discrepancies. Finally, 1.07% of crashes occurred when `llhook` failed to load the extracted native library, likely because `llhook` was unable to overwrite the original `System.loadLibrary()`.

Thus, the main limitations addressable to StegoPack are due to the merger component. In particular, the conflict resolution will likely fail if both host and payload use the `AppCompat` dependency [63]. Even in an empty app, using `AppCompat` automatically generates around 850 Android XML resource files for more than 32,000 default resource entires that will deterministically collide [64] . Our algorithm struggles to propagate new identifiers and names for each resource (plus the one effectively defined in the two apps) in all the other resource files that refer to them.

Discussion on functional test outcomes. The 33% success rate in executing the stegomalware may initially seem like a low success rate. However, it indicates that, in one-third of the cases, simply selecting two random apps results in successful execution of the stegomalware. This demonstrates the feasibility of the Stegopack model. While building a universal tool to ensure functional integration between any host and any malware in the wild proved impractical—primarily due to the need for manual handling of corner cases—this challenge is more theoretical than practical. The actual goal of the attacker is not to take random malware and pair it with random hosts. Instead, the attacker seeks to inject dex/so payloads containing only the malicious logic into a carefully selected, suitable host app. As such, it is not essential for the payload to function seamlessly on every host, but rather to be successfully embedded and executed within a host that supports the payload with minimal adjustments.

Regarding the antivirus assessment, we evaluated the entire *Stegomalware Dataset*, including apps that were not runnable. This decision was made because the malware's functionality on the victim's device is not necessary for evaluating its ability to evade detection by antivirus engines. Since the static features of the stegopacked APK remain unchanged regardless of its runtime behavior, the malware's execution on the device does not affect its ability to stay undetected by AV analysis.

5.4. Malware packing capacity

This analysis aims to empirically learn how much "*packing space*" real-world apps typically provide and whether this is sufficient for typical malware payloads. We analyzed all the 13,195 malware from our dataset. Table 2 presents payload size statistics for 13,195 `dex` malware instances (note that the `classes.dex` file is mandatory for native malware) and 2316 native malware samples.

Average file sizes are not meaningful as the dataset includes ready-to-run malware samples incorporating facade logic to appear as legal software. Consequently, `dex` and `so` files, including layout and visual component dependencies, lead to larger file sizes. Our primary concern is the malicious logic that attackers embed in these samples. Thus, we focus on the impact of samples composed mainly of code, specifically those targeting ransomware, remote access trojans (RATs), lockers, spyware, and Metasploit-based payloads. We chose the 75th percentile as a reference point for our analysis, as it provides a realistic upper bound that accommodates most observed samples, capturing typical cases rather than outliers.

The results show that 75% of the `dex` files are below 901.7 KB. Larger sizes are due to dependencies or external libraries. Even potent payloads, like Metasploit's reverse shells, typically range from 20 KB to 100 KB. For native malware, encoding shared object (.so) files is necessary, and sizes can reach several MB. About 75% of payloads are under 1.13 MB. These smaller payloads align with our scenarios of attackers modifying existing apps, being the legitimate facade provided by the host app.

Table 2
File size statistics for `dex` and `so` files.

Statistic	DEX	SO
Number of Samples	13,195	2316
Minimum Size	4.6 KB	2.4 KB
Maximum Size	25.1 MB	100.51 MB
Average Size	838.7 KB	4.59 MB
25th Percentile	47.6 KB	20.92 KB
50th Percentile	159.8 KB	336.98 KB
75th Percentile	901.7 KB	1.13 MB
90th Percentile	2.18 MB	13.13 MB

Table 3
Apps' pixels and embedding capacity.

Category	Assets (*percentage*)	Average Pixels (*M pixels*)	Max Payload (*M bytes*)
Banking	40.7%	15.25 MP	3.81 MB
Social	53.7%	23.63 MP	5.90 MB
Gaming	45.9%	56.22 MP	14.05 MB

Thus, considering the 75th percentile as a reference for `dex` and `so` files, host apps need an embedding capacity of up to 2 MB (combined `dex` and `so` parts). Assuming a steganographic algorithm operating at *2 bits per pixel*, 8 million pixels of image assets are required to hide such assets. While this embedding capacity may seem limiting, it is important to note that the actual payloads in typical malware often fall within these size ranges. Moreover, if the host app lacks space to embed larger payloads, additional images can be added to this extent.

Thus, considering the 75th percentile as a reference for `dex` and `so` files, host apps need an embedding capacity of up to 2 MB (combined `dex` and `so` parts). Assuming a steganographic algorithm operating at *2 bits per pixel*, approximately 8 million pixels of image assets are required to hide such payloads. While this embedding capacity may seem limiting at first glance, it is crucial to note that typical malware payloads often fall within these size ranges. Furthermore, if the host app lacks sufficient space to accommodate larger payloads, additional image assets can be used to extend the embedding capacity.

To determine whether apps typically have this capacity and to identify suitable types for embedding malicious payloads, Table 3 presents an analysis conducted on three categories from the *Goodware Dataset*: *54 social apps*, *61 gaming apps*, and *59 banking apps*.

For each category, we report the percentage of apps that carry uncompressed image assets (column "Asset" - assets packaged in compressed files were excluded from this analysis) and the average number of pixels available in PNG, JPG, SVG, and BMP formats.

Results show that *if* the app provides assets, the embedding capacity is mainly sufficient. Gaming apps have significantly more image assets than social or banking apps, making them more suitable for embedding larger payloads. However, in all categories, half or more of the apps *do not utilize uncompressed image assets*, implying that careful consideration is needed to identify hosts with effective payload embedding capacity.

5.5. Impact of payload and host choice

To systematically establish the impact of payload and host selection on detection results, we submitted all 14,000 stegopacked payload–host pairs to VirusTotal and evaluated each pair against 79 antivirus engines. This analysis yielded three key insights.

- **Significant reduction in detection rate**: The average detection rate drastically dropped after repackaging (quantitative details in the following);
- **Impact of the host app**: As shown in Fig. 11, the detection rate almost deterministically (!) depends on whether the host app is known or unknown; known hosts are widely recognized apps on the Google Play Store, while unknown hosts include lesser-known Play Store apps and unpublished apps, such as those developed in-house, downloaded from GitHub, or sourced from alternative stores. Using a known app as the host reduced the detection rate from 39.85% to an already very significant 4.5% (in almost all cases, 500 detections out of 11,060 tests), but using an unknown host lowers it further to as low as 0.4% (45 detections out of 11,060 tests).[4] The difference in detection rates for known hosts can be attributed to AVs profiling code fingerprints of well-known apps, allowing them to spot deviations introduced after repackaging (see Section 5.7);
- **Consistency within categories**: Apart from the distinction between known and unknown hosts, the specific choice of host does not affect detection rates, meaning detection outcomes remain consistent within the same host category. This suggests that current antivirus solutions do not employ steganalysis techniques, as embedding rates and image variations do not influence detection (see Section 6).

Since the specific choice of the host does not influence results beyond being known or unknown, we focus the remainder of this section on comparing two apps: a well-known app, `com.gamma.scan` (a basic barcode scanner with over 500 million

[4] 11,060 tests refer to 140 malware samples repackaged within each host, scanned against 79 AVs.

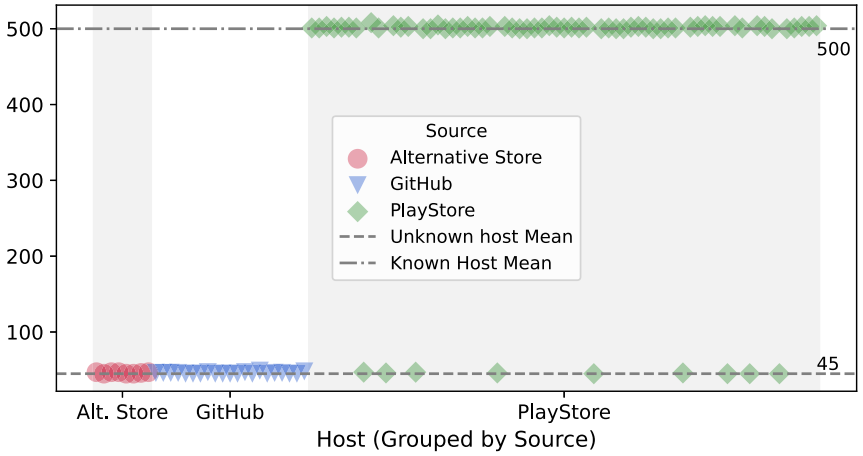


Fig. 11. Scatterplot showing the total number of detections for all stegopacked payloads within each host app, categorized by the host's source.

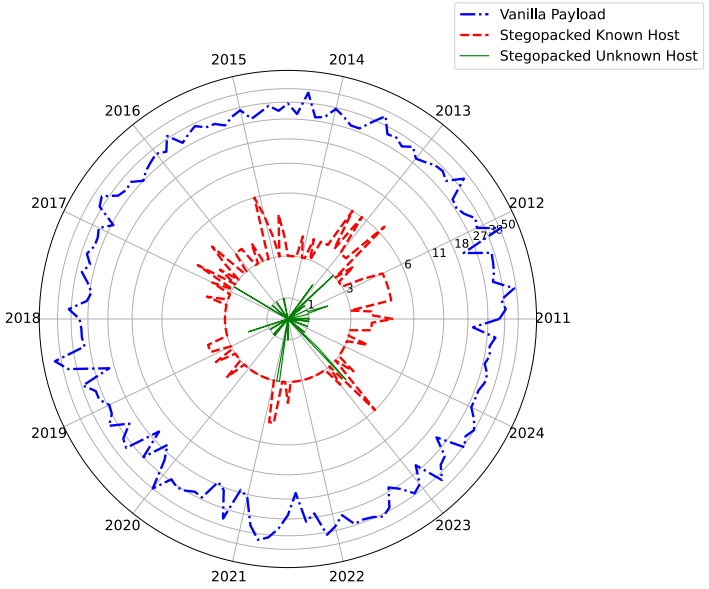


Fig. 12. Total detection comparison (log scale) between unpacked payloads embedded in known and unknown hosts.

Table 4
Detection results for malware and benign payloads in known and unknown hosts across different packing stages.

Scenario	Original sample	Modified resources	Stego loader	Repack. app
Malware/Known Host	40/79	40/79	0/79	**3**/79
Malware/Unknown Host	40/79	40/79	0/79	0/79
Benign/Known Host	0/79	0/79	0/79	**3**/79
Benign/Unknown Host	0/79	0/79	0/79	0/79

downloads on the Play Store), and an app developed by us (`it.runningexamples.fiscalcode`) as the unknown host. The radar plot in Fig. 12 compares the detection results by repackaging each payload within known and unknown hosts. The plot reveals a striking anomaly: regardless of the malware variant used, known apps consistently yield at least three detections compared to unknown hosts.

For further insight, we focus on a specific malware instance (SmsSpy 2021), initially detected by 40 of the 79 AVs, as a use case. The first two lines in Table 4 present detection results for such malware, when packed into known and unknown hosts, across four stages: original sample, modified sample after merging malware and app resources, sample after loading the malware into the

Table 5
AV Detection effectiveness on 140 StegoPacked malware samples using the unknown app as host.

AV	# Detections (%)	Of which detected as same threat
BitdefenderFalx	19 (13.57%)	11 (7.8%)
K7GW	8 (5.7%)	7 (5.0%)
Kaspersky	4 (2.8%)	4 (2.8%)
ZoneAlarm	4 (2.8%)	4 (2.8%)
Sophos	2 (1.4%)	0 (0%)
ClamAV	1 (0.7%)	1 (0.7%)

app resources, and final repackaged app. As expected, the detection results for the original malware and the modified resources are identical since the merger component has a functional rather than an evasive role. In particular, the steganographic Loader's detection rate drops to zero, indicating no antivirus software can detect the concealed payload in the images.

Previous results suggest that current antivirus solutions cannot detect payloads within assets and label the StegoPack steganography extraction process as malicious. After repackaging, three antivirus programs—ESET-NOD32, Ikarus, and Google—detect the malware again. However, such detections differ from those on the original payload and are independent of the malware used (as confirmed with other malware samples). Initially, ESET-NOD32 labeled the SmsSpy malware as a variant of `Android/Spy.Banker.AYT` while Ikarus reported `Trojan.AndroidOS.SmsSpy` (Google, in all cases, returns a generic `detected` response).

After repackaging, all classifications could not recognize the type of malware correctly. ESET indeed detected an `Android/TrojanDropper.Agent.LGT`, while Ikarus recognized the repackaged app as `Trojan.AndroidOS.Obfus`.

The reason for this wrong detection is intuitively explained by looking at the description of the `TrojanDropper` threat [65]:

> *Android/Trojan.Dropper is a malicious app that contains additional malicious app(s) within its payload. This infected APK typically is given the filename of a legitimate app but has a completely different package name, digital certificate, and code than the app it claims.*

In other words, detection seems based on identifying deviations in the app's composition compared to its expected structure rather than applying more advanced heuristics specifically designed to detect stealth alterations of assets, such as StegoPack's steganographic manipulation. To verify this hypothesis, we repeated the analysis with a benign payload (a simple app that displays a Toast message). As shown in line 3 of Table 4, even for a payload initially labeled as non-malicious, the repackaging process itself triggers detections when using a known app as the host, further supporting the conclusion that such detections rely on static analysis of modifications in well-known apps—indeed, no detection occurs in the case of unknown apps.

5.6. AV detection effectiveness

Here, we evaluate the effectiveness of different antivirus software in detecting malware-embedded in-app assets through steganography. We focus on repackaging using an unknown host, as detection is more challenging in this case and aligns with our threat model.

Table 5 summarizes the detection results. Of the 79 AVs tested, only the eight listed in the table successfully identified some threats, despite the concealment provided by steganography.[5] The tables' rightmost column reports whether the threat was detected as the original malware. As discussed in the previous section, detection may label the threat differently from the unpacked version of the malware sample. Out of 45 total detection instances, the AVs identified the original threat in 27 cases. The best-performing antivirus is *BitDefenderFalx*, which detected 19 out of 140 malware samples, with 11 of those detections correctly identifying the same threat as the original malware.

To further understand how much the detection rate is related to the malware's seniority, Fig. 13 shows detections on payloads, categorized by each year. Despite the payloads being embedded steganographically, consistent detection across AVs suggests that the detection mechanisms are likely analyzing malware, focusing on the *dex* file and other APK metadata. Specifically, while the steganography stage shows zero detections (see Table 4), the AVs can identify malicious payloads even after repackaging. This fact implies that the detection is based on analyzing the payload's inherent characteristics, such as its original strings and assets, which may be revealed or merged during the embedding process. If there is a signature in the manifest or resources, these are embedded as-is since we focused on concealing the `dex` file and related code.

Through manual analysis, we found that the 2011–2013 payloads detections are due to the presence of `Lotoor` malware variants. These malware contain malicious ELF files within their asset folders, designed to exploit older Android vulnerabilities to gain root access to devices [66]. The detection is triggered by these malicious components stored raw in the assets, which are embedded in the repackaged malware during the merging phase. In particular, Lotoor exploits have been mitigated since the Gingerbread release,

[5] We excluded Avast-Mobile since it labels both vanilla and stegopacked malwares with generic "`Android:Evo-gen [Tr]`", often associated with goodware apps (https://forum.avast.com/index.php?topic=321616.msg1695001#msg1695001, https://forum.avast.com/index.php?topic=323096.0). F-Secure and BitDefender have one false positive each (Vanilla undetected/StegoPacked detected).

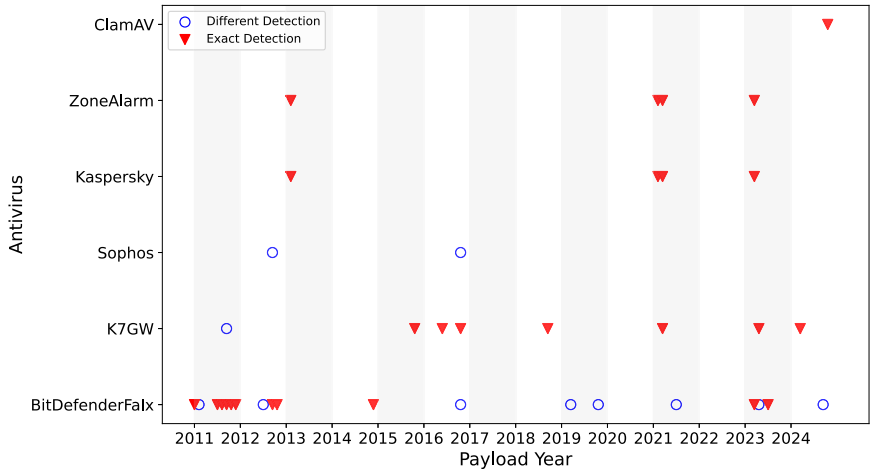


Fig. 13. AVs performance in detecting the same threat of the original payload.

making them obsolete threats [67,68]. Malware that utilizes raw malicious resources falls outside the scope of StegoPack, which is designed to embed malware using steganography for stealth strategically. However, such malware constitutes significant outliers. Of the total 4464 detections recorded for the original malware, only 27 was consistent with the original threat, which is a mere 0.67% of effective detections.

5.7. Evading ESET detection

The tests in Section 5.5 revealed that only three of the 79 antivirus programs evaluated (ESET, Ikarus, and Google) consistently detected our approach when applied to widely recognized apps. This detection reliability likely stems from their use of an incremental scanning approach [69], which focuses on examining new components within an app. When an app's name or package is recognized, these AVs compare the submitted app to a known benign version, identifying code changes and control flow graph patterns indicative of threats. In our case, ESET (correctly!) flagged repackaging functionalities as `TrojanDropper`.

To understand the detection mechanisms and whether they are robust enough to thwart the tool's evasion methods, we focused on ESET and examined which part was detected. We thus conducted a comprehensive analysis by progressively removing sections of the Loader and performing thorough scans. The results indicated that the `InMemoryDexClassLoader` API was considered an IoC. Removing the code segment employing this API (refer to Appendix D, Listing 1) resulted in no detection. Although this makes the malicious app non-operational, it confirms that ESET identifies the `InMemoryDexClassLoader` API as malicious.

At this stage, a natural question was whether it was possible to preserve the malicious functionalities while making the repackaged app elusive. To achieve this, a trivial solution was to try disrupting detection by inserting an additional `invoke-static` call between the `InMemoryDexClassLoader` constructor and the `getClassLoader()` API (Appendix D, Listing 2). This was sufficient to evade ESET detection, although Ikarus and Google continued to detect the StegoPack loader, but only when injected into known apps.

5.8. Repeating experiments over time

The results presented above date back to April 2024, with 45 detections for the unknown app and 500 for the known one. When repeated in June 2024 using the same apps, the detections increased significantly to 489 for the unknown app and 1002 for the known one. In particular, the number of initially detected threats also increased (131 for the unknown host and 304 for the known ones), probably due to a deeper manual analysis allowing AVs to identify the original malware in some cases. Kaspersky and ZoneAlarm consistently flagged many of the generated malware instances as `TrojanDropper`, explicitly pinpointing the steganographic loader stage. ESET similarly identified (again) instances as `TrojanDropper`, including those associated with previously unknown apps, suggesting that these apps have become recognizable to specific AV engines over time.

We slightly modified the structure of the steganographic loader and performed experiments to assess the effectiveness of AV countermeasures against our attack. We refactored the loader, renaming project elements (assets, packages, classes, methods, variables) while preserving semantic integrity to circumvent Kaspersky and ZoneAlarm. Following recompilation and scanning, we received zero detections from these AVs. Regarding ESET, we adopted additional code-scrambling techniques, following the strategy already outlined in Section 5.7.

Repackaging the malware within a previously used known app (`com.speedsoftware.explorer`) and submitting 140 samples to VirusTotal led to a decrease in detections, dropping to 169, with 20 related to the original malware threats. This experiment suggests that, while some AV engines eventually flag originally submitted apps as malicious, they mainly rely on signatures without fully understanding the steganographic packing methods used in the attack. Moreover, they have not developed any behavioral signatures capable of detecting our attack beyond the steganographic loader's facade.

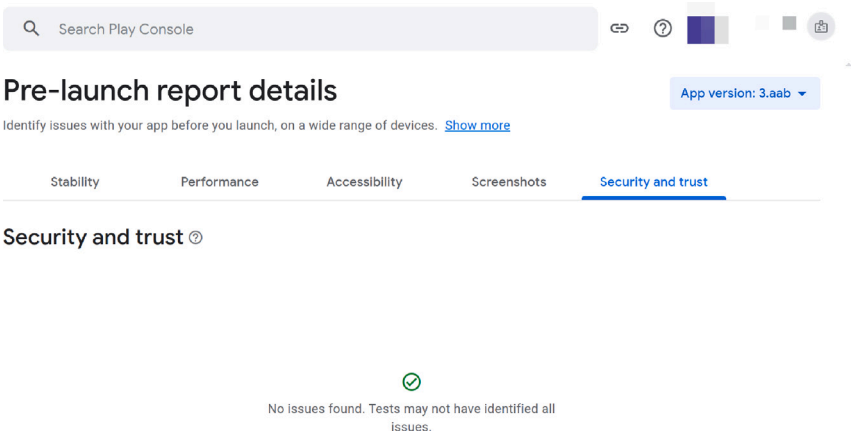


Fig. 14. Play Console's initial review phase cleared without security issues, allowing beta testing.

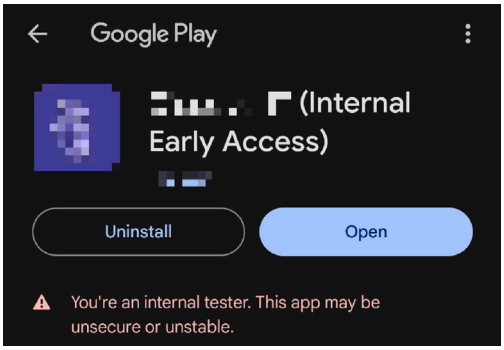


Fig. 15. Closed beta release on Play Store.

5.9. Play store submission

Following our threat model, we tried to demonstrate the feasibility of distributing stegomalware on the Google Play Store. To this aim, we selected a malware sample containing a Metasploit reverse shell. The sample has been flagged as malicious by 26 AVs, including Google, in its original form. We used StegoPack to embed the malware sample in the host app. Then, we submitted the malicious host app to the Google Play Store.[6]

The malicious host app successfully passed Play Console security checks (pre-launch report, Fig. 14) and is currently in the "Closed Beta" phase (Fig. 15), available only to selected testers and not publicly accessible. This phase represents a significant milestone, as attackers could exploit it by sending invitation links to potential victims, allowing them to install the malware directly from the Play Store. Although the malicious host app could be publicly released after 14 days of beta testing, we have kept it private for ethical reasons and will not proceed with a public release. We further verified that the app bypasses *Play Protect's real-time detection*,[7] granting the installation of the malicious host app as a legitimate one.

Figs. 14, 15, 16 illustrate the successful upload of the malicious host app up to the closed beta level. Specifically, Fig. 14 details the pre-launch report, which granted the app closed beta status, and Fig. 15 shows the app on the private Google Play, accessible to explicitly invited beta testers. Fig. 16 shows the run-time Analysis of Play Protect output.

6. Detection strategies

The analysis shows that none of the 79 AVs evaluated via VirusTotal seems to account for an adversary that steganographically embeds malware in the app assets. However, as shown in the previous sections, the integration of steganographic approaches into the overall structure of the app and the dynamics of payload execution is a viable direction. Even intentionally basic steganographic

[6] DISCLAIMER: The embedded malicious payload is completely harmless, as it attempts to contact a non-existent private IP, thus *never activates*, even when extracted.

[7] https://security.googleblog.com/2023/10/enhanced-google-play-protect-real-time.html

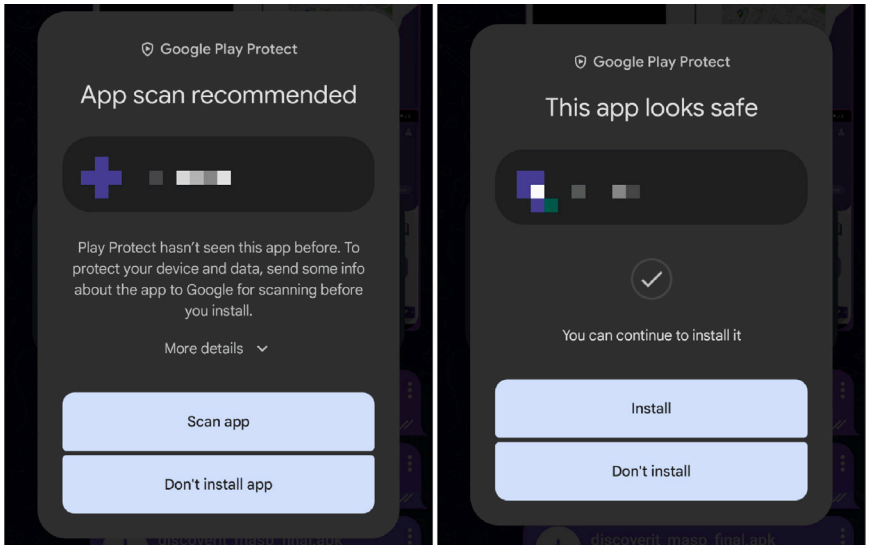


Fig. 16. Successful circumvention of the Play Protect runtime analysis.

techniques, as used in this paper, pose significant challenges to traditional detection mechanisms; indeed, no AV engines successfully identified our payload as malicious during testing. We could address defense by detecting the loading phase, as, for instance, done by ESET. However, as discussed in Section 5.7, stealth adversaries may easily bypass simplistic techniques to detect suspicious control flows. The presence of `InMemoryDexClassLoader` alone is not a sufficient IoC because it is frequently used for legitimate purposes within the Android ecosystem (see Section 6.4). Therefore, this API should be considered dangerous only when used with other suspicious activities. The key to effective defense lies in *fighting steganographic adversaries with steganography detectors*, which should be carefully integrated with techniques designed to recognize strategies for retrieving encoded data from images and dynamic code loading.

6.1. Towards detecting stegomalware based on stegopack

The first step is to recognize the strategies implemented in the app code for retrieving encoded data from images. Although we can detect the use of our specific algorithm, it is essential to develop more general solutions that can identify any potential encoding algorithm that might be employed. Some basic strategies are described below, along with the relevant limitations.

Stegdetect & Suárez-Tangil's approach [19]**.** Devised explicitly for Android stegomalware, this approach identifies suspicious apps capable of executing dynamic code. The assets of these flagged apps are then analyzed using the Stegdetect tool [70]. However, this strategy has several notable limitations. First, flagging native code as suspicious is ineffective since modern apps commonly use native libraries. Moreover, Stegdetect is designed for known steganographic techniques within a single file, whereas our methodology disperses information across multiple images with varying embedding rates, which it cannot handle. Most importantly, Stegdetect lacks dynamic engines to reconstruct the malware payload. Relying solely on static analysis can lead to a non-negligible number of false positives, as steganography might be legitimately used to encode sensitive information.

Static Signatures on Loader Component. During the assessment of AV detection, a key finding emerged: AVs primarily rely on static signatures that target the specific syntax and methods defined in the steganographic loader rather than employing behavioral signatures to capture the overall workflow of stegopacked malware. We demonstrated that simple modifications to the loader code are sufficient to avoid these signatures, highlighting a significant gap in developing efficient, instruction-agnostic detection approaches. In this sense, a viable detection method could involve recognizing the LSB decoding process, typically achieved by shifting all bits and performing a bitwise `AND` operation with `0x1`. However, the extensive use of shift operations in benign APKs may lead to false positives, making it challenging to distinguish between malicious and benign uses.

Towards an Holistic Detection Approach. Considering the abovementioned limitations, we developed a more generalized detection method to handle the broader and more complex scenarios posed by modern stegomalware. To address these challenges, we follow a *hybrid analysis* approach that integrates both static and dynamic analysis features of the malware:

1. Static Analysis: Identify `DexClassLoader` or `InMemoryDexClassLoader` usage as indicators of potential dynamic code loading;
2. Steganalysis: Analyze app assets to detect steganographic manipulation, assessing whether images may have been altered to embed hidden data, regardless of the specific stego-method employed;

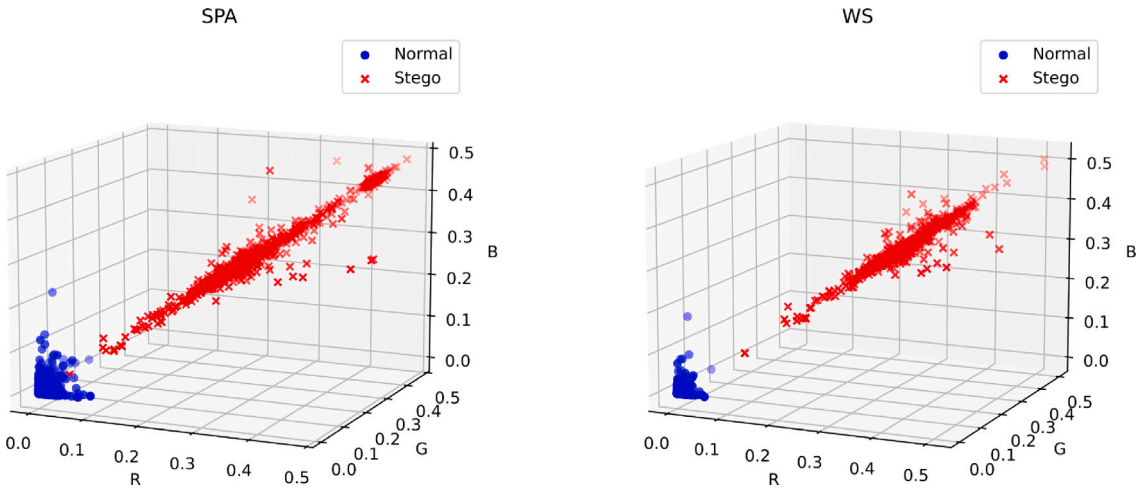


Fig. 17. Distribution of WS and SPA analysis results on R, G, B components.

Table 6
Mean RGB values for SPA and WS methods for low embedding rates.

Embedding rate	SPA	SPA Stego	WS	WS Stego
0.02bpp	0.0045	0.0063	0.0007	0.0033
0.04bpp	0.0052	0.0117	0.0008	0.0108
0.06bpp	0.0067	0.0127	0.0013	0.0130
0.08bpp	0.0063	0.0197	0.0010	0.0220

3. Dynamic Hooking: Implement dynamic hooking to intercept the buffer provided to `InMemoryDexClassLoader` and the full path of the loaded native library. This enables real-time reconstruction and analysis of the Dex and `so` files as they are dynamically loaded, facilitating immediate monitoring of the code execution;
4. Malware Verification: Submit the reconstructed dex and so files to detection services (e.g., VirusTotal) to check for malicious code.

6.2. Assessment of steganalysis methods

We apply the methods described in the proposed approach to detect steganographically embedded content in the APK's assets. It is worth mentioning that, in this phase of steganography detection, the baseline goal is not to retrieve the original payload but to identify the potential presence of stegomalware. Once detected, analysts can study the decoding algorithm, allowing the hidden payload to be reconstructed statically.

To do this, we used the following two methods: SPA (Sample Pair Analysis) [17] and WS (Weighted Steganography) [71]. The first analyzes pairs of samples to detect the presence of steganographic modifications. It examines statistical properties such as correlations between adjacent pixels to identify alterations suggesting hidden content. The latter identifies hidden payloads by evaluating the distortion introduced by steganographic embedding, assigning different weights to pixels based on their significance.

Preliminary Investigation. We initially tested the validity of these methods by encoding 250 MB of `dex` files within one GB of steganographic images at the maximum embedding rate of 2 bpp. Applying both steganalysis methods to the original and encoded images, we compared their results for each color component. Fig. 17 shows a clear and easily separable clustering between the original and stego images. Both steganalysis techniques yield significantly higher coefficients across all color components in manipulated images.

Variable Embedding Rate. Stegomalware can encode payloads at variable embedding rates (see Appendix A). We analyze the impact of our algorithm by varying the embedding rate as the accuracy of the steganalysis decreases with a lower percentage of encoded pixels [17]. Since StegoPack equally loads the three color channels, Fig. 18 shows the average values for the R, G, and B channels in both the original and stego images for varying embedding rates up to the maximum of $2bpp$. The results show that even with relatively low embedding rates, there is a noticeable distinction between images with and without steganography. However, embedding rates can drop to as low as 0.02 bpp and 0.08 bpp for apps rich in assets such as gaming and messaging, respectively. From Table 6, it can be seen that for low embedding rates, the distinction between original and steganographic images is less pronounced, suggesting that setting a threshold for detecting embedding rates lower than 0.1 bpp could result in false positives. This fact implies that a safe threshold must be at least 0.025, which catches a minimum embedding rate of 0.1 bpp.

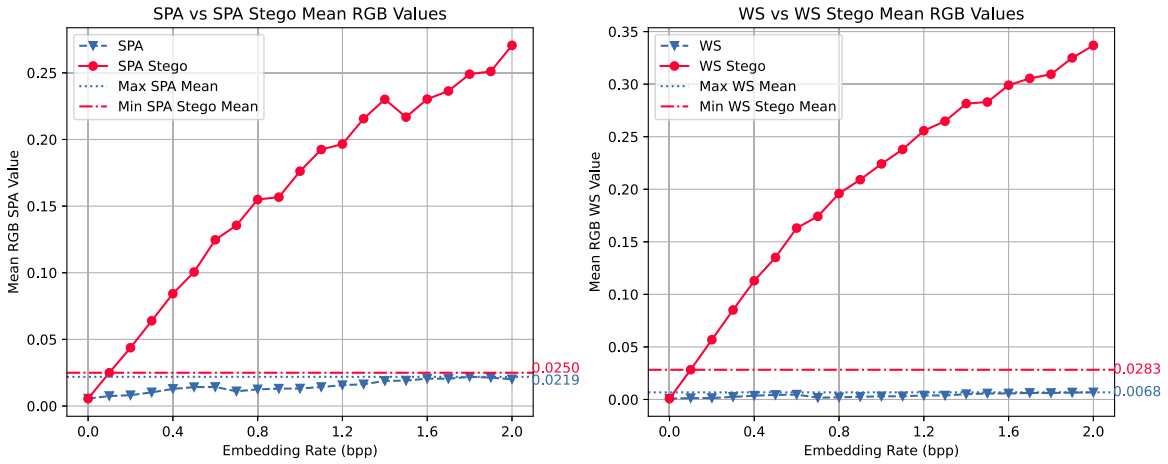


Fig. 18. Steganalysis compares steganographic images and their original version at varying embedding rates.

Table 7
Real APK images: analysis by color channel.

Method	Channel	Mean	25th perc.	50th perc.	75th perc.
SPA	R	0.02630	0.00000	0.00844	0.03245
	G	0.02786	0.00000	0.00923	0.03531
	B	0.02781	0.00000	0.01070	0.03531
WS	R	0.02794	0.00000	0.00000	0.00696
	G	0.02679	0.00000	0.00000	0.00660
	B	0.02689	0.00000	0.00000	0.00653

Threshold Establishment via Analysis of Legitimate APKs. To determine a reliable threshold for steganalysis and avoid false positives, we analyzed the results of steganalysis on 274 existing legitimate APKs. Upon examination of 33,281 images from these apps, we found that the mean values in Table 7 are above the previously established minimum threshold of 0.025, while the median values are lower. This indicates that certain assets can trick the steganalysis (see Appendix C). In the distribution shown in Fig. 23, 4470 of the 33,281 images exhibit values above the 0.025 threshold, resulting in a false positive rate of 13.43%. Increasing the threshold to 0.05, 2988 out of 33,281 images would exceed the threshold, lowering the false positive rate to 8.98% but reducing detection to embedding rates around 0.2 bpp or more.

Since SPA and WS have distinct limitations based on image characteristics (see Appendix C), we suggest developing defensive strategies that combine both methods for each color component in detection policies.[8] However, SPA and WS struggle with small and monochromatic images, which can lead to false positives in steganalysis. This indicates that steganalysis alone is insufficient for accurate stegomalware detection and should be supplemented with other domain-specific IOCs.

6.3. Dynamic payload reconstruction

After identifying a sufficient number of suspicious assets with the potential to embed a payload and confirming the presence of dynamic code loading capabilities, the next task is reconstructing the `dex` and `so` files of the embedded payload. We utilize Frida [72] to execute a script to intercept and connect to API calls for dynamic code loading. For Java components, our Frida script (see Appendix D, Listings 3) hooks into the `InMemoryDexClassLoader` method. This allows monitoring and capturing the `ByteBuffer` parameter from memory. By dumping the contents of the ByteBuffer, we can reconstruct the `dex` file of the payload. For native components, the Frida script (see Appendix D, Listings 4) hooks into the `System.load()` invocation. It captures the library path argument provided to this API, reads the specified library file, and then copies it to a local directory for further analysis. Having them in their original form, the reconstructed `dex` and `so` files are subsequently analyzed using VirusTotal to detect the presence of malicious code.

[8] For instance, being $R_{SPA}, G_{SPA}, B_{SPA}$ and R_{WS}, G_{WS}, B_{WS} denote the results of steganalysis indicators, and being thr_{SPA} and thr_{WS} detection thresholds for the SPA and WS method, respectively, trigger an alarm when $(R_{SPA} \vee G_{SPA} \vee B_{SPA} > thr_{SPA})$ and/or $(R_{WS} \vee G_{WS} \vee B_{WS} > thr_{WS})$. Note that detection policies should account for the possibility that some steganographic algorithms might encode data selectively in one or more color components while ignoring others.

6.4. Goodware evaluation

To assess the effectiveness and precision of our detection tool, we evaluated the 274 benign apps from the Goodware Dataset (including the 100 hosts used in the Stegomalware Dataset). The results of this evaluation are summarized as follows:

1. Dynamic Code Loading and Steganalysis: Of the 274 goodware apps, 25 were flagged for dynamic code loading and potential steganographic manipulation. Static analysis would have marked these as false positives. Dynamically reconstructing the associated payload files with dynamic hooking would confirm their benign nature, thus mitigating static analysis false positives.
2. Dynamic Code Loading Only: 87 apps exhibited dynamic code loading capabilities without any steganographic indications, confirming that dynamic code loading is prevalent among legitimate apps and alone does not suffice as an indicator of malicious intentions.
3. Steganalysis Only: 20 apps exhibited signs of steganographic manipulation but did not utilize dynamic code loading. These apps may contain steganographic data in the images, but the steganalysis algorithms could also generate false positives for certain assets (as discussed in Section 6.2). However, these apps reasonably do not embed a malicious payload, as they lack dynamic code-loading capabilities.

Globally, 45 apps triggered false positives in the steganalysis: 15 gaming apps, 8 banking apps, 16 social apps, 4 apps for image/video editing, and 2 generic tools. The initial false positives identified by static analysis and steganalysis were resolved by dynamic hooking and subsequent verification of the `so` and `dex` files with antivirus scans. Thus, our methodology ensures that the final assessment has no false positives. By applying the proposed hybrid analysis mechanism, we effectively mitigate the risk of false alarms while maintaining high accuracy in distinguishing between benign and malicious apps.

6.5. StegoPacked malware evaluation

We evaluated the entire *Stegomalware Dataset* to assess the effectiveness of our hybrid detection mechanism against StegoPack. Among the 45 apps from the *Goodware Dataset* that triggered steganalysis, 12 were used as hosts in the stegomalware generation. To ensure a fair evaluation, we excluded the 1680 samples derived from these hosts, as their deterministic false positives made them unsuitable choices for hosting stegomalware.

We analyzed the remaining 12,320 stegopacked apps with our detector. The results show that 12,140 of these apps were detected accurately as stegomalware, achieving a *success rate of 98.53%*. The remaining 180 apps, which evaded steganalysis, are wrongly classified as goodware. In particular, 169 of these apps had a poor embedding rate (below 0.2 bpp), which aligns with our chosen threshold of 0.05 - a compromise to balance sensitivity to low embedding rates and minimize false positives. Furthermore, 11 of the evading apps had embedding rates between 0.21 bpp and 0.35 bpp. This variability is due to differences in the sizes and types of assets within different host apps, which affect their response to steganalysis.

For further validation, we evaluated the VirusTotal detection rate for all `dex` files, which were consistently dumped at runtime from stegopacked malware. To also reliably include reconstructed `so` files, this approach should be supplemented with dynamic techniques for a broader test coverage.[9] A comparison between the detection rates of the original malware APKs and their corresponding `dex` files is illustrated in Fig. 19. Although the detection rate for payload files (2920 detections) was lower than that for the original APKs (4408 detections), it is crucial that most AV solutions still recognize each `dex` file. The reduction in detection rate is attributed to the fact that some IOCs considered by AV engines are located outside `classes.dex` or within native libraries (e.g., `liblib<name>.so` files). As a result, these IOCs can only be detected by scanning the full APK rather than individual components.

The results demonstrate the robustness of our detection mechanism in identifying most stego-packed payloads, showcasing its effectiveness against this attack vector.

However, the approach still has **notable limitations**. On the *dynamic side*, malware may not activate immediately after the startup; instead, it waits for specific user interactions. This delayed activation, coupled with the limited observation period typical of dynamic analysis, can lead to cases where the effective payload is neither reconstructed nor detected.

On the *steganalysis side*, we observed the presence of false positives, as 45 applications from the *Goodware Dataset* triggered steganalysis. Additionally, when payloads are minimal or embedded in hosts with substantial asset bases, low embedding rates can enable some payloads to evade detection. Finally, the steganographic threshold we employed was calibrated based on the characteristics of our specific algorithm and may not directly generalize to other steganographic schemes.

These challenges underscore the complexity of countering such advanced attacks, particularly when attackers strategically pair malware payloads with host apps to evade detection. Although our approach demonstrates promise for strengthening the Android ecosystem against these threats, more work is required to address these nuanced attack strategies more comprehensively.

[9] In our testing environment, the `dex` file is consistently retrieved as it is loaded at the application's startup. However, for `so` files, we must ensure that malicious code delivered reaches the point where `loadLibrary()` is invoked.

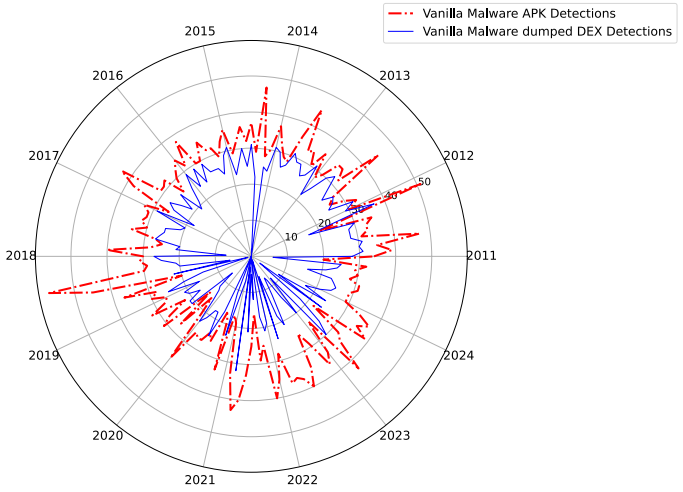


Fig. 19. Detection results: comparison between original malware APKs and their extracted `dex` files.

7. Conclusions and future works

This paper presented a methodology applied to `dex` or native apps to hide arbitrary malicious payloads within the assets of Android apps using steganography. Despite deliberately employing basic steganographic techniques, our experimental analysis revealed significant deficiencies in current AV solutions and automated Play Protect assessments; notably, we successfully uploaded stegopacked malware to a closed beta level without detection. None of the detection engines tested identified the embedded payloads as malicious. In the rare cases where detection occurred, the heuristics were not related to malware embedding within app images. This suggests that most AV solutions' threat models do not consider steganographic adversaries.

Our findings further indicate that steganalysis alone is insufficient in the context of Android apps due to potential false positives. We, therefore, advocate for advanced detection approaches that focus on app assets to uncover hidden threats. We also complement static steganalysis and signature-based detection with dynamic hooking techniques to monitor app behavior at runtime. This allows for the reconstruction of the code dynamically loaded by the malware. However, we argue that such remediation is just a first step, and further research is required to design more advanced detection solutions.

A potential limitation of stegomalware is that counter-steganography techniques could disrupt it. However, applying such defenses in Android is non-trivial. First, preprocessing techniques such as adding noise to images must be unpredictable to prevent stegomalware from adapting and encoding payloads resistant to these transformations. Second, suppose stego images are stored within the *assets* folder of an APK. In that case, they remain read-only at runtime, preventing modification through preprocessing mechanisms operating on the installed app. This implies that countermeasures would need to intervene before installation. Modifying a compiled APK also invalidates the original developer signature, requiring the app to be re-signed with a different keystore. This would break the integrity guarantees of Android's signature verification model.

Considering these challenges, future work will focus on developing robust and scalable countermeasures that disrupt stegomalware without impacting legitimate applications. Addressing these technical limitations will be key to designing effective mitigation strategies against steganographic threats.

CRediT authorship contribution statement

Danilo Dell'Orco: Writing – original draft, Software, Methodology, Formal analysis, Conceptualization. **Giorgio Bernardinetti:** Methodology, Investigation. **Giuseppe Bianchi:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Funding acquisition. **Alessio Merlo:** Writing – review & editing, Writing – original draft, Supervision, Methodology. **Alessandro Pellegrini:** Writing – original draft, Resources, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Steganography encoding algorithm

The steganography encoding algorithm currently used in StegoPack uses a simple Least Significant Bit (LSB) method with a maximum embedding capacity of 2 bits per pixel (bpp) [73]. We iterate over payload bytes and image pixels, modifying only 2 of the three color components using a round-robin algorithm based on the positional index of the bit pair within the current byte.

Fig. 20. Original Image, Image After Steganography Embedding, and Modified Pixels Highlighted in Red.
Source: FFHQ Dataset: https://www.kaggle.com/datasets/arnaud58/flickrfaceshq-dataset-ffhq.

Table 8
PSNR statistics.

Statistic	Value (dB)
Min PSNR	52.19
25th Percentile	54.07
Median PSNR	55.80
Mean PSNR	56.77
75th Percentile	58.64
Max PSNR	68.65

Visual imperceptibility. In the worst-case scenario (from a stealth perspective) of two bits per pixel, we evaluated the visual imperceptibility of the steganography algorithm used on a set of 1000 images of different sizes. We evaluated the Peak Signal-to-Noise Ratio (PSNR) [74] to measure the loss of quality between the original and steganographic images. Considering that a PSNR of 30 dB is generally considered reasonable in terms of image quality for steganography applications [75], the results shown in Table 8 confirm the minimal loss of visual image quality. The example image shown in Fig. 20 highlights how the original and steganographic images appear indistinguishable to the human eye. However, nearly all pixels had their color components altered by a $+/-1$ value.

Variable payload embedding rate. Our encoding strategy distributes the payload bytes across all available assets, resulting in a variable embedding rate for each payload–host pair, depending on payload size and the total number of pixels within the host application's images. Let P_{req} denote the number of pixels required to encode N bytes of payload. Considering that every pixel may contain up to two bits, $P_{req} = N$ [bytes] $\times 8$ [bits/byte]$/2$ [bits/pixel] $= 4N$ [pixels]. Being P_{tot} the total number of pixels available in the host application's assets, the percentage α of bits to be encoded in each image is $\alpha = P_{req}/P_{tot} \in [0, 1]$, i.e., the embedding rate per-image is $E_i = 2\alpha$ bits per pixel. Practical embedding was then performed by stego-encoding, in addition to the malware payload and a control header specifying how bits are distributed across images.

Appendix B. Custom packing format for native libraries

Compared to the `classes.dex` file, native libraries can be numerous, each compiled for a specific CPU architecture. To handle native libraries, we have therefore constructed a structured packet to encapsulate all the library data, as depicted in Fig. 21. The preamble contains crucial information for library reconstruction during the decoding phase. It specifies the library name, supported CPU architectures, and the size of each `.so` file. This enables StegoPack to reconstruct the original libraries after extraction from the stego images accurately.

Appendix C. False positive detection of legitimate APK images

Most of the assets that exceed the detection thresholds come from game applications characterized by small textures with monochromatic or patterned designs. Examples are shown in Fig. 22. The size of the image is the main factor that affects these results, as shown in Fig. 24. This situation is expected, as the SPA and WS metrics are significantly influenced by image characteristics, including compression methods and preprocessing techniques [76,77]; individual pixels have a more significant impact on smaller images. For images sized 256×256, the values deviate from the decreasing trend due to 238 resources from a specific gaming app, which consists mainly of white font images. Additionally, images with random or patterned visual elements often show a higher

Preamble											
4 bytes	4 bytes	4 bytes	var Bytes		4 bytes	var Bytes	4 bytes	var Bytes		4 bytes	var Bytes
#archs	#libs	arch1_name_s	arch1_name	...	archN_name_s	archN_name	lib1_name_s	lib1_name	...	libK_name_s	libK_name

Data						
4 bytes	var Bytes	4 bytes	var Bytes		4 bytes	var Bytes
arc1lib1_data_s	arc1lib1_data	arc1lib2_data_s	arc1lib2_data	...	arc1libK_data_s	arc1libK_data
4 bytes	var Bytes	4 bytes	var Bytes		4 bytes	var Bytes
arc2lib1_data_s	arc2lib1_data	arc2lib2_data_s	arc2lib2_data	...	arc2libK_data_s	arc2libK_data
var Bytes						
...						
4 bytes	var Bytes	4 bytes	var Bytes		4 bytes	var Bytes
arcNlib1_data_s	arcNlib1_data	arcNlib2_data_s	arcNlib2_data	...	arcNlibK_data_s	arcNlibK_data

Fig. 21. Native libraries packet encoding format.

Fig. 22. Example of images fooling steganalysis.

pixel correlation, which is associated with structured data and is a common trait in steganographic content. Overall, from these results, we empirically observe that:

1. SPA results increase with decreasing image size, even without steganography.
2. WS is more stable, as shown by the percentage plot (Fig. 18). However, it degrades faster for tiny images concerning SPA. It returns a probability near 1 for $16x16$ images or smaller, with a transparency channel and only a few colors. This is typical for texture assets for games.
3. SPA and WS methods show higher stego probabilities for small images (lower than 64×64 pixels).
4. SPA gives values between 0.1 and 0.25 for images with random or constant visual patterns or noise filters. WS appears to be more robust in these cases.
5. Both SPA and WS yield values between 0.1 and 0.2 for images that are predominantly monochrome on a specific color component.

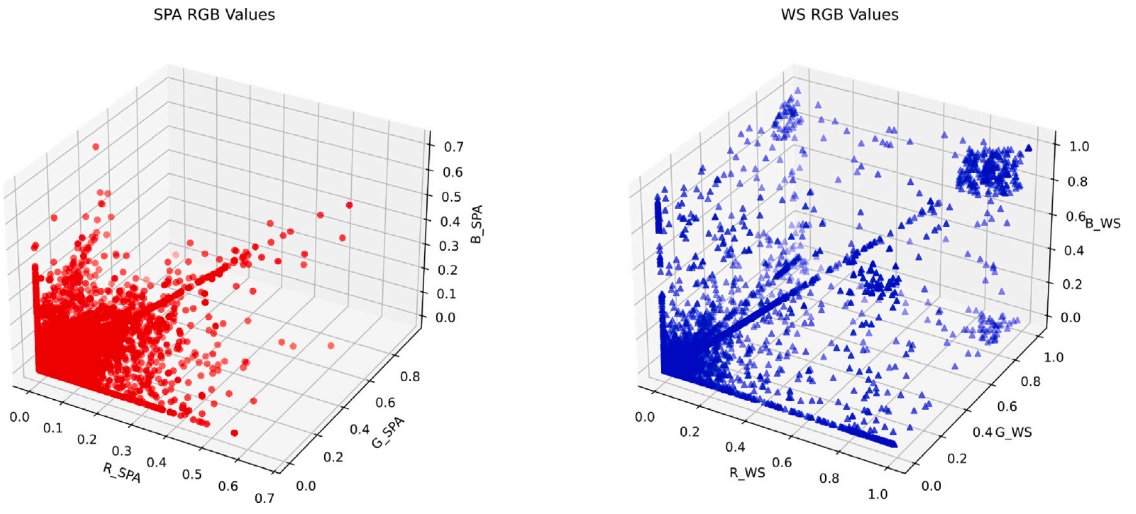


Fig. 23. SPA and WS steganalysis results on goodware raw assets.

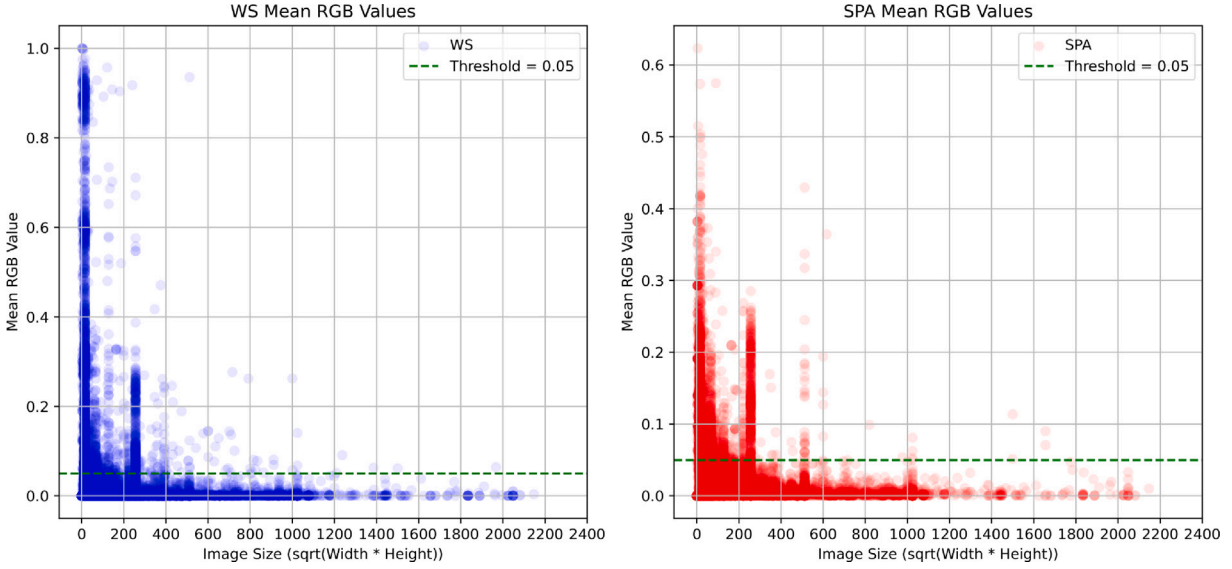


Fig. 24. Steganalysis on goodware assets about image size.

Appendix D. Code artifacts

```
new-instance v1, Ldalvik/system/InMemoryDexClassLoader;

invoke-virtual {p0}, Landroid/content/Context;->getClassLoader()Ljava/lang/ClassLoader;
move-result-object v2

invoke-direct {v1, v0, v2}, Ldalvik/system/InMemoryDexClassLoader;-><init>(Ljava/nio/ByteBuffer;
    Ljava/lang/ClassLoader;)V
```

Listing 1 Code snippet detected as malicious by ESET, utilizing the `InMemoryDexClassLoader` API

```
.method public doRegisterOperations(I)I
.registers 4
move v0, p1
```

```
const/4 v1, 5
add-int v1, v0, v1
const/4 v2, 3
mul-int v2, v1, v2
return v2
.end method

...

invoke-virtual {p0}, Landroid/content/Context;->getClassLoader()Ljava/lang/ClassLoader;
move-result-object v2

const/4 v0, 7
invoke-virtual {p0, v0}, Lcom/tools/render_library/MainActivity;->doRegisterOperations(I)I

invoke-direct {v1, v0, v2}, Ldalvik/system/InMemoryDexClassLoader;-><init>(Ljava/nio/ByteBuffer;
    Ljava/lang/ClassLoader;)V
```

Listing 2 Calling of the fake method to disrupt the Control Flow Graph

```
function InMemoryDexClassLoader_dump() {
var memoryclassLoader = Java.use("dalvik.system.InMemoryDexClassLoader");
memoryclassLoader.$init.overload('java.nio.ByteBuffer', 'java.lang.ClassLoader').implementation =
    function (dexbuffer, loader) {
    const dump_file = `/data/data/${PACKAGE_NAME}/dump.dex`;
    var original_classloader = this.$init(dexbuffer, loader); // Mandatory. After read dexbuffer
    will be empty

    // Reading Data from ByteBuffer provided to InMemoryDexClassLoader
    var remaining = dexbuffer.remaining();
    var buf = new Uint8Array(remaining);
    for (var i = 0; i < remaining; i++) {
        buf[i] = dexbuffer.get();
    }
    console.log("[*] Read Completed of " + remaining + " bytes")

    // We dump the dex in /data/data app private folder instead. Cannot write to /sdcard unless the
    app has rights to.
    const f = new File(dump_file, 'wb');
    f.write(buf);
    f.close();
    return original_classloader;
}
```

Listing 3 Frida script to hook the `InMemoryDexClassLoader` API, and reconstruct the original Malware DEX file.

```
function NativeLibraryLoader_dump() {
    Java.perform(function () {
        // Hook the System.load() method
        var System = Java.use('java.lang.System');
        var File = Java.use('java.io.File');
        var FileInputStream = Java.use('java.io.FileInputStream');
        var FileOutputStream = Java.use('java.io.FileOutputStream');

        System.load.overload('java.lang.String').implementation = function (path) {
            console.log("[*] System.load() called with path: " + path);

            const dump_file = `/data/data/${PACKAGE_NAME}/dump_library.so`;

            var file = File.$new(path);
            if (file.exists()) {
                var fileSize = parseInt(file.length());
                console.log("[*] File Size: " + fileSize);
                var buf = new Uint8Array(fileSize);

                var inputStream = FileInputStream.$new(file);
                inputStream.read(buf);

                // Write the library to the dump file
                var dump = File.$new(dump_file);
```

```
                var outputStream = FileOutputStream.$new(dump);
                outputStream.write(buf);
                console.log("[*] Library dumped to: " + dump_file);
                inputStream.close();
                outputStream.close();

            } else {
                console.error("[*] File does not exist: " + path);
            }
        }

        return this.load(path);
    });
}
```

Listing 4 Frida script to hook the `load()` API, and reconstruct the original Malware SO file.

Data availability

Data will be made available on request.

References

[1] V. Kouliaridis, K. Barmpatsalou, G. Kambourakis, S. Chen, A survey on mobile malware detection techniques, IEICE Trans. Inf. Syst. 103 (2) (2020) 204–211.

[2] N.F. Johnson, S. Jajodia, Exploring steganography: Seeing the unseen, Computer 31 (2) (1998) 26–34.

[3] 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.

[4] 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, in: ESEC/FSE 2022, Association for Computing Machinery, New York, NY, USA, 2022, pp. 1430–1440.

[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] ProGuard, Guardsquare, [Online]. Available: https://www.guardsquare.com/proguard.

[7] U. Nawaz, M. Aleem, J.C.-W. Lin, On the evaluation of android malware detectors against code-obfuscation techniques, PeerJ Comput. Sci. 8 (2022) e1002.

[8] M. Hammad, J. Garcia, S. Malek, A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 421–431.

[9] X. Zhang, F. Breitinger, E. Luechinger, S. O'Shaughnessy, Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations, Forensic Sci. Int.: Digit. Investig. 39 (2021).

[10] Z. Muhammad, M.F. Amjad, H. Abbas, Z. Iqbal, A. Azhar, A. Yasin, H. Iesar, A systematic evaluation of android anti-malware tools for detection of contemporary malware, in: 2021 IEEE 19th International Conference on Embedded and Ubiquitous Computing, EUC, 2021, pp. 117–124.

[11] K. Tam, A. Feizollah, N.B. Anuar, R. Salleh, L. Cavallaro, The evolution of android malware and android analysis techniques, ACM Comput. Surv. 49 (4) (2017) 1–41.

[12] L. Caviglione, W. Mazurczyk, Never mind the malware, here's the stegomalware, IEEE Secur. Priv. 20 (5) (2022) 101–106.

[13] G. Bernardinetti, D. Di Cristofaro, G. Bianchi, PEzoNG: Advanced packer for automated evasion on windows, J. Comput. Virol. Hacking Tech. 18 (4) (2022) 315–331.

[14] J. Fridrich, M. Goljan, R. Du, Reliable detection of LSB steganography in color and grayscale images, Proc. the 2001 Work. Multimed. Secur.: New Challenges (2002).

[15] T. Zhang, X. Ping, A new approach to reliable detection of LSB steganography in natural images, Signal Process. 83 (10) (2003) 2085–2093.

[16] O. Dabeer, K. Sullivan, U. Madhow, S. Chandrasekaran, B. Manjunath, Detection of hiding in the least significant bit, IEEE Trans. Signal Process. 52 (10) (2004) 3046–3058.

[17] S. Dumitrescu, X. Wu, Z. Wang, Detection of LSB steganography via sample pair analysis, in: Information Hiding: 5th International Workshop, IH 2002 Noordwijkerhout, the Netherlands, October 7–9, 2002 Revised Papers 5, Springer, 2003, pp. 355–372.

[18] S. Ge, Y. Gao, R. Wang, Least significant bit steganography detection with machine learning techniques, in: Proceedings of the 2007 International Workshop on Domain Driven Data Mining, 2007, pp. 24–32.

[19] 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.

[20] S. Badhani, S. Muttoo, Evading android anti-malware by hiding malicious application inside images, Int. J. Syst. Assur. Eng. Manag. 9 (2017).

[21] 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, in: A-Mobile 2018, Association for Computing Machinery, New York, NY, USA, 2018, pp. 25–28.

[22] Baidu APK Protect, Baidu Inc., [Online]. Available: http://apkprotect.baidu.com/.

[23] Bangcle, Bangcle Inc., [Online]. Available: http://www.bangcle.com/.

[24] Ijiami, Ijiami Inc., [Online]. Available: http://www.ijiami.cn/.

[25] DexGuard: State-of-the-art Protection for Android apps and SDKs, Guardsquare, 2023, [Online]. Available: https://www.guardsquare.com/dexguard. (Accessed: 2023-11-28).

[26] Reversing an Android app Protector: Part 2 - Assets and Code Encryption - JEB in Action, PNF Software, [Online]. Available: https://www.pnfsoftware.com/blog/reversing-android-protector-encryption/.
[27] R. Fedler, M. Kulicke, J. Schütte, An antivirus API for android malware recognition, in: 2013 8th International Conference on Malicious and Unwanted Software: "the Americas", MALWARE, 2013, pp. 77–84.
[28] S. Tanner, I. Vogels, R. Wattenhofer, Protecting android apps from repackaging using native code, in: A. Benzekri, M. Barbeau, G. Gong, R. Laborde, J. Garcia-Alfaro (Eds.), Foundations and Practice of Security, Springer International Publishing, Cham, 2020, pp. 189–204.
[29] VirusTotal: Free Online Virus, Malware and URL Scanner, VirusTotal, [Online]. Available: https://www.virustotal.com/.
[30] Google, APK structure, 2023, [Online]. Available: https://developer.android.com/topic/performance/reduce-apk-size#apk-structure. Specifically, the APK structure.
[31] R. Winsniewski, Android–apktool: A tool for reverse engineering android apk files, 2012, [Online]. Available: https://apktool.org/.
[32] Y. Zhou, X. Jiang, Dissecting android malware: Characterization and evolution, in: 2012 IEEE Symposium on Security and Privacy, IEEE, 2012, pp. 95–109.
[33] S. Rastogi, K. Bhushan, B. Gupta, Measuring android app repackaging prevalence based on the permissions of app, Procedia Technol. 24 (2016) 1436–1444.
[34] Y. Ishii, T. Watanabe, M. Akiyama, T. Mori, Appraiser: A large scale analysis of android clone apps, IEICE Trans. Inf. Syst. 100 (8) (2017) 1703–1713.
[35] A. Merlo, A. Ruggia, L. Sciolla, L. Verderame, You shall not repackage! demystifying anti-repackaging on android, Comput. Secur. 103 (2021).
[36] A. Merlo, A. Ruggia, L. Sciolla, L. Verderame, ARMAND: Anti-repackaging through multi-pattern anti-tampering based on native detection, Pervasive Mob. Comput. 76 (2021) 101443.
[37] 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.
[38] 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.
[39] RedNaga, APKiD, https://github.com/rednaga/APKiD, ongoing.
[40] 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.
[41] Quick Heal, Q3 2020 threat report, 2020, https://www.quickheal.co.in/documents/threat-report/qh-threat-report-q3-2020.pdf.
[42] FortiGuard Encyclopedia: Virus - ID 8283330, FortiGuard Labs, [Online]. Available: https://www.fortiguard.com/encyclopedia/virus/8283330.
[43] FortiGuard Encyclopedia: Virus - ID 7962728, FortiGuard Labs, [Online]. Available: https://www.fortiguard.com/encyclopedia/virus/7962728.
[44] FortiGuard Encyclopedia: Virus - ID 7866616, FortiGuard Labs, [Online]. Available: https://fortiguard.fortinet.com/encyclopedia/virus/7866616.
[45] C. Gao, M. Cai, S. Yin, G. Huang, H. Li, W. Yuan, X. Luo, Obfuscation-resilient android malware analysis based on complementary features, IEEE Trans. Inf. Forensics Secur. 18 (2023) 5056–5068.
[46] O. Mirzaei, J. de Fuentes, J. Tapiador, L. Gonzalez-Manzano, AndrODet: An adaptive android obfuscation detector, Future Gener. Comput. Syst. 90 (2019) 240–261.
[47] 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, ARES '18, Association for Computing Machinery, New York, NY, USA, 2018, [Online]. Available: http://dx.doi.org/10.1145/3230833.3232823.
[48] G. Suarez-Tangil, S.K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, L. Cavallaro, Droidsieve: Fast and accurate classification of obfuscated android malware, in: Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy, 2017, pp. 309–320.
[49] C. Sun, H. Zhang, S. Qin, J. Qin, Y. Shi, Q. Wen, Droidpdf: The obfuscation resilient packer detection framework for android apps, IEEE Access 8 (2020) 167460–167474.
[50] P. Graux, J.-F. Lalande, V.V.T. Tong, Obfuscated android application development, in: Proceedings of the Third Central European Cybersecurity Conference, 2019, pp. 1–6.
[51] T. Morkel, J.H. Eloff, M.S. Olivier, An overview of image steganography., in: ISSA, vol. 1, (2) 2005, pp. 1–11.
[52] A. Carrega, L. Caviglione, M. Repetto, M. Zuppelli, Programmable data gathering for detecting stegomalware, in: Proceedings of the 6th IEEE Conference on Network Softwarization, 2020, pp. 422–429.
[53] A. Martín, A. Hernández, M. Alazab, J. Jung, D. Camacho, Evolving generative adversarial networks to improve image steganography, Expert Syst. Appl. 222 (2023) 119841.
[54] A. Aggarwal, A. Sangal, A. Varshney, Image steganography using LSB algorithm, in: International Journal of Information Sciences and Application, IJISA, vol. 11, (1) International Research Publication House, 2019.
[55] M. Spreitzenbarth, F. Freiling, Android malware on the rise, Tech. Rep., (CS-2012-04) University of Erlangen, Dept. of Computer Science, 2012.
[56] S. Gupta, A. Goyal, B. Bhushan, Information hiding using least significant bit steganography and cryptography, Int. J. Mod. Educ. Comput. Sci. 4 (6) (2012) 27.
[57] A. Ruggia, A. Possemato, S. Dambra, A. Merlo, S. Aonzo, D. Balzarotti, The dark side of native code on android, 2023, http://dx.doi.org/10.36227/techrxiv.21220247.v1, [Online]. Available: http://dx.doi.org/10.36227/techrxiv.21220247.v1.
[58] Android Developers, Android System.loadLibrary() API, 2024, [Online]. Available: https://developer.android.com/reference/java/lang/System#loadLibrary(java.lang.String). (Accessed: 2024-06-05).
[59] Android Developers, Android System.load() API, 2024, [Online]. Available: https://developer.android.com/reference/java/lang/System#load(java.lang.String). (Accessed: 2024-06-05).
[60] Android Open Source Project, Linker namespace, 2024, [Online]. Available: https://source.android.com/docs/core/architecture/vndk/linker-namespace. (Accessed: 2024-06-05).
[61] L. Kirk, Artful, 2023, [Online]. Available: https://github.com/LaurieWired/ARTful.
[62] L. Kirk, Runtime riddles: Abusing manipulation points in the android source, in: DEF CON 31, 2023, [Online]. Available: https://media.defcon.org/DEF%20CON%2031/DEF%20CON%31%20presentations/Laurie%20Kirk%20-%20Runtime%20Riddles%20Abusing%20Manipulation%20Points%20in%20the%20Android%20Source.pdf.
[63] Android Developers, AndroidX AppCompat, 2025, https://developer.android.com/jetpack/androidx/releases/appcompat. (Accessed: 2025-03-04).
[64] Google, Material components for android, 2024, https://github.com/material-components/material-components-android/tree/master/lib/java/com/google/android/material/resources/res. (Accessed: 2024-10-30).
[65] Malwarebytes, Trojan Dropper Malware Threat, MalwareBytes, 2023, [Online]. Available: https://www.malwarebytes.com/blog/detections/android-trojan-dropper.
[66] Kaspersky Lab, Exploit.AndroidOS.Lotoor, Kaspersky Lab, 2021, [Online]. Available: https://threats.kaspersky.com/en/threat/Exploit.AndroidOS.Lotoor/.
[67] T. Armstrong, Malware in the android market, 2011, [Online]. Available: https://securelist.com/malware-in-the-android-market/29834/.
[68] D. Maslennikov, Malware in the android market, part 2, 2011, [Online]. Available: https://www.securelist.com/malware-in-the-android-market-part-2/160552/.

[69] HCLSoftware, HCL AppScan: Incremental scan of an application, 2023, [Online]. Available: https://help.hcltechsw.com/appscan/Standard/10.3.0/c_IncrementalScans.html.
[70] O.S. Khalind, J.C. Hernandez-Castro, B. Aziz, A study on the false positive rate of stegdetect, Digit. Investig. 9 (3–4) (2013) 235–245.
[71] X. Yu, N. Babaguchi, Weighted stego-image based steganalysis in multiple least significant bits, in: 2008 IEEE International Conference on Multimedia and Expo, 2008, pp. 265–268.
[72] Frida - A World-Class Dynamic Instrumentation Framework, Frida Project, 2024, [Online]. Available: https://frida.re/docs/android/. (Accessed: 2024-07-10).
[73] R. Crandall, Some notes on steganography, Posted Steganography Mail. List. 1998 (1) (1998) 6.
[74] Peak signal-to-noise ratio, 2024, [Online]. Available: https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio. (Accessed: 2024-05-31).
[75] D.R.I.M. Setiadi, PSNR vs SSIM: imperceptibility quality assessment for image steganography, Multimedia Tools Appl. 80 (6) (2021) 8423–8444.
[76] A.D. Ker, Quantitative evaluation of pairs and RS steganalysis, in: Security, Steganography, and Watermarking of Multimedia Contents VI, vol. 5306, SPIE, 2004, pp. 83–97.
[77] A.D. Ker, R. Böhme, Revisiting weighted stego-image steganalysis, in: Security, Forensics, Steganography, and Watermarking of Multimedia Contents X, 6819, SPIE, 2008, pp. 56–72.