Forgotten Libraries: Assessing Systemic Android Insecurity

Luca Di Bartolomeo, Philipp Mao, Mathias Payer
EPFL

***Abstract*—Android app developers bundle native libraries, but often neglect to update them. Attackers can leverage known memory corruption vulnerabilities in these native libraries to remotely compromise apps. In a case study, researchers demonstrated the "outdatedness" of libraries across a few hundred Android apps using heuristics to identify the library versions.**

Inspired—and worried—by this case study, we perform a massive, systematic analysis of the entire Android native ecosystem. We analyze native libraries from every app in the AndroZoo dataset, totaling over 8 million apps. By building a ground-truth dataset of library releases and distribution artifacts, we identify the exact version for 65% of the shipped libraries, relying on deterministic supply-chain matches.

Our results indicate a more severe state of Android native libraries than previously assumed. In a deep dive, we select five popular native libraries that were affected by several vulnerabilities in the past. Among the 66,545 apps that bundle these libraries, 79.9% still include a vulnerable version from at least 4 years ago in their latest release. We manually verify the exploitability of 300 of those apps, confirming that an attacker can effectively trigger the vulnerability in at least 34 of them. Vulnerable apps include banking apps and cryptocurrency wallets. These applications are all currently available on the Play Store, with some boasting hundreds of millions of downloads.

I. Introduction

While Android apps mix many different languages and environments, performance-critical code is generally shipped as standalone native libraries. Native code lacks many of the security guarantees that the Dalvik Virtual Machine provides. The ubiquity of native libraries in the Android ecosystem makes them a ripe target for analysis and exploitation. A single vulnerability in a popular native library might lead to countless exploitable applications.

Prior work [1], [2] performed a preliminary analysis of a subset of apps, identifying that 26% of the 200 most popular Android apps relied on native libraries with known vulnerabilities. While OSSPolice required source code and heavyweight processing, Librarian relied on pattern matching to identify library versions. Both struggled with inaccuracy. Accurately pinpointing the version of the libraries used by apps is highly complex. Binary similarity on a broad scale is notoriously difficult [3] and the sheer scale of the Android ecosystem poses a significant challenge. Neither study provided a thorough analysis of the exploitability of these vulnerabilities, failing to assess the effective risk to the end user.

To overcome this barrier, we leverage domain-specific information to directly match libraries by building an extensive ground-truth dataset. Restricting our focus to *only* Android native libraries allows us to include all Play Store apps in our analysis and significantly improves the accuracy. We do not require expensive hardware or imprecise heuristics, and we can accurately identify over 65% of the libraries used by apps in the entire Android ecosystem.

The results of our analysis reveal that the state of the ecosystem is considerably worse than previously thought. Even when focusing only on highly critical and longstanding CVEs, the number of vulnerable apps is staggering. This includes apps with over a hundred million downloads. For example, we find that 58.91% of the 5,790 apps on the Play Store using `libpl_droidsonroids_gif.so` remain vulnerable to CVE-2019-11932. Other libraries present an even worse picture: 99.34% of the 16,688 apps using `libpdfium.so` are vulnerable to CVE-2018-6120. All of the 11,794 apps relying on `libucrop.so` are vulnerable to CVE-2018-7641.

Additionally, to measure the real risk posed to end users as a result of vulnerabilities in widespread native libraries, we conduct a manual evaluation of their exploitability. Among a sample of Play Store apps, we confirm remote crash/exploit triggers in at least 34, including banking apps and crypto wallets.

The key observation for our analysis is that developers download or compile native libraries from a small number of sources. The popularity of these library sources enables us to build a ground-truth library dataset by querying each source and collecting metadata for every single native library available. After collecting extensive metadata, we can then catalog libraries used by apps to perform a comprehensive analysis of the state of Android native libraries. This straightforward solution for library identification significantly outperforms previous approaches in performance, scale, and accuracy.

Through our ground-truth dataset, we examine *all* apps collected by AndroZoo (more than 8 million apps from different markets) for a complete and systematic overview of the entire Android ecosystem. Analyzing such a large number of apps enables us to conduct a complete supply-chain study, detecting if an app or package copied native libraries from another one, and recursively tracking down their true origin.

Network and Distributed System Security (NDSS) Symposium 2027
22 – 26 March 2027, Seoul, Republic of Korea
ISBN 978-1-970672-09-1
https://dx.doi.org/10.14722/ndss.2027.230745
www.ndss-symposium.org

Ultimately, this work provides the following contributions and takeaways:

- **Supply-chain deterministic matching.** We match app-shipped binaries to exact versions from ground-truth sources, identifying versions for over 65% of library usages.
- **Ecosystem-scale measurement.** Our approach scales to all latest-version apps on AndroZoo (including the Play Store), totaling 8,914,212 apps. Contrary to the results highlighted by previous work (which estimated 26% of native-library-bundling apps are vulnerable), our deep-dive into 66,545 apps utilizing five widespread libraries indicates that 79.9% of these apps remain vulnerable to critical, years-old CVEs.
- **Reachability in the wild.** We manually assess reachability and confirm remotely triggerable memory corruptions in at least 34 out of 300 Play Store apps using vulnerable libraries, demonstrating real user risk.
- **Open source.** We publish our code and (anonymized) dataset at https://anonymous.4open.science/r/acid-C468/.

Our work provides a clear signal that this is a systemic issue, not limited to a few apps. We conclude by discussing potential causes for this and proposing actionable changes to mitigate the problem.

II. Our Dataset

One key objective of this study is to address the lack of a comprehensive ground-truth dataset for native libraries used in Android. We first analyze the usage of native libraries in Android applications. Then, we collect ground-truth information about native libraries from eight different sources.

A. AndroZoo

We download apps from AndroZoo [4], a collection of Android applications from various app markets, such as the Google Play Store, F-Droid, and AppChina. AndroZoo hosts more than 25 million APK files, keeping track of every version ever published of each application. We only consider the latest version of each application, reducing the total number of apps considered in this study to 8,914,212. The apps were downloaded between August 1 and August 8, 2025.

From each APK, we extract the binaries inside the folder `/lib/arm64-v8a/`. Note that while Play Store guidelines require using this folder, some apps may try to circumvent this. We tested on a sample of apps and did not find any instance of this circumvention (see Section VI). Parsing a single folder allows us to severely reduce bandwidth requirements as explained in Appendix A. This results in 1,077,144 apps bundling native libraries in their latest version.

AndroZoo includes old apps that have since been removed from the Play Store. For the purposes of our study, we are interested in the apps that might be vulnerable but are still installable on an Android device by an average user. Since 2023, AndroZoo provides Google Play Store metadata for each app. Unfortunately, we found that the Play Store metadata was incomplete. We thus resort to scraping metadata directly from the Play Store ourselves.

B. Play Store Metadata

Play Store metadata indicates the number of installations of each app, the date of the last update, detailed developer information, and most importantly whether the app is still available on the Play Store.

We noticed that the AndroZoo Play Store metadata is unreliable, as more than half of the apps are not actually available on the Play Store anymore. To solve this issue, we rely on the Python package `google-play-scraper` [5] to query the Play Store for each individual app and retrieve the relevant metadata.

We query the Play Store for each of the 1,077,144 apps that contained native libraries in our dataset. Out of those, we verify that 186,944 are still available on the Play Store and we retrieve their relevant metadata.

C. Ground Truth for Native Libraries

We collect the most popular sources of Android native libraries used by developers.

To determine the most popular sources, we sort the libraries obtained in Section II-A by prevalence. We then investigate the origin of the most prevalent libraries included in at least 15,000 apps.

For each source, we write a Python scraper that downloads all hosted native libraries, calculates their hash, and stores all the available metadata that each source provides. Our primary focus is on collecting data regarding the official library name, its version, and the release date. We identify 8 different sources:

- **Maven Central:** The most popular source, Maven Central distributes a large number of Java and Android packages. Native libraries are included in `.aar` archives, and are accompanied by metadata contained in `.pom` XML files.
- **Jitpack:** Similar to Maven Central, Jitpack is a Maven repository that ships `.aar` packages and their metadata. Due to its first-class GitHub support, Jitpack is more popular among newer libraries.
- **npm:** The Node Package Manager is responsible for delivering the native libraries for React Native (the most popular Android native library) for all the versions released from 2016 to 2022.
- **Flutter:** Flutter is a framework to develop cross-platform mobile apps. The native libraries of the framework are shipped by Google in zip archives.
- **Android NDK:** Google's official releases of the Android Native Development Kit framework contain many popular libraries like the C++ standard library that apps often include.
- **React-Glog** and **React-Folly:** React also provides the Glog logging library and Facebook's Folly C++ runtime.
- **Renderscript:** A computation engine to perform computationally expensive tasks such as image manipulation.

Google ships the prebuilt native libraries as part of their AOSP repository.

Each of these sources required a custom approach when scraping their native libraries and parsing the provided metadata. A detailed explanation of our approach for each of them can be found in Appendix B.

III. Analysis

We now correlate libraries shipped in Android apps with the ones we collected from the ground-truth sources.

A. Matching Library Hashes

When matching libraries from our dataset with the ones collected from the sources, we compare the SHA256 hash of the binary. This simple approach works for most libraries, but there are some limitations. First, two developers compiling the same library might end up with different binaries due to variations in development environments. Second, some libraries include the build timestamp in the binary, which also would produce a different hash. A number of techniques exist to address this issue by designing custom hashing algorithms resistant to small changes that do not modify the behavior of the binary (called "fuzzy hashing" [3]). However, those techniques are not suitable for our purposes as they are heuristic and cannot guarantee the correctness of the match.

We partially mitigate this issue by relying on a conservative approach: in our SHA256 hash we only include three sections of a binary: the code section (`.text`) and the data sections (`.data` and `.rodata`). We thus ignore sections such as `.comment`, `.note`, and `.gnu.hash` that have no effect on the behavior of the library. With this method, we can match 10% more libraries compared to hashing the full binary.

This approach may lead to false negatives (we may miss two libraries that should be matched) but will not lead to false positives (two different libraries will not be matched). We consider special adversarial strategies (e.g., renaming the `.text` section) out of scope for this study.

a) Different compiler versions, flags and patches: Prior work on Android native library identification aims to ignore subtle differences in the code (e.g., compiler flags, small patches).

However, for the purposes of our study, we take the opposite approach. Different compiler flags may lead to different vulnerabilities being present in the library. Small patches are even more problematic as they might be backported security updates. Our approach instead will not match libraries compiled with different compiler flags or with any patches applied. In fact, any of those would lead to changes in the `.text` sections and thus different hashes. At the same time, this property prevents false vulnerability claims: if a vendor backports a security patch, its hash will change and we will not match it to the vulnerable version, and the app will not be counted as vulnerable.

B. Discerning the True Origin of a Library

Maven, Jitpack, npm, and other distribution sources ship native libraries as part of *packages*. While some libraries only appear in a single specific package, and clearly originate from that, there are cases where the same binary appears in multiple different packages. The cause of this is that some packages do not use other packages as dependencies (e.g., in the case of Maven, they are not listed as Maven dependencies in their `.pom` metadata file) but rather they just directly include the package "copy-pasted" from somewhere else. In these cases the metadata from the including package cannot be relied upon to report accurate versions of the copied package.

To address this issue and discover a library's origin, we use the package publishing timestamps. If the same library is shipped by two different packages, we mark only the oldest package as the true source for this library. This step is not sufficient, however: some open-source libraries are custom compiled by the package developers and then included in the package. We take a conservative approach to mitigate this: we consider a package to be the true source of a library if and only if that package distributed the largest number of libraries with that specific name. In fact, it is fairly common to find libraries such as `libucrop.so` where there is one package shipping a single version and the official package shipping four (for each major release). In that case, we consider the one with four versions to be the official source.

We manually evaluate the accuracy of our ground truth against open-source apps taken from F-Droid. Section IV-B details our approach. In our representative sample of 645 libraries, our ground truth is accurate in 91.7% of cases.

IV. Studying the Android Library Ecosystem

Our analysis addresses the following questions:

- **RQ1:** How complete is the ground truth data we gathered? How many libraries are we potentially missing?
- **RQ2:** How trustworthy is the ground truth data we gathered?
- **RQ3:** How outdated are native libraries in the Android ecosystem?
- **RQ4:** How many apps are shipping vulnerable native libraries?
- **RQ5:** Are libraries not matched from our ground truth dataset outdated as well?
- **RQ6:** Are the affected apps exploitable in a real-world scenario?

We now go over each question explaining our methodology to address it. Afterward, we provide a discussion of the results we obtained.

A. Completeness of Ground Truth (RQ1)

We analyze the completeness of our dataset by looking at how many apps use libraries we do not recognize. Table II and Figure 1 give an overview of the popularity of the different sources where Android developers get native libraries from. The "Unknown" category represents libraries not present in our dataset. Note that Maven Central is by far the most popular

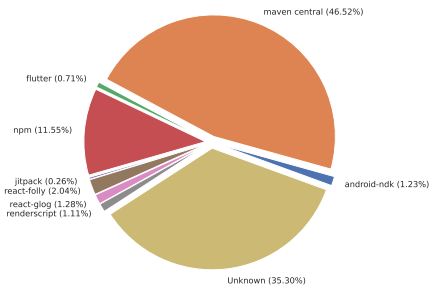


Fig. 1: Pie chart of the different sources where Android native libraries come from.

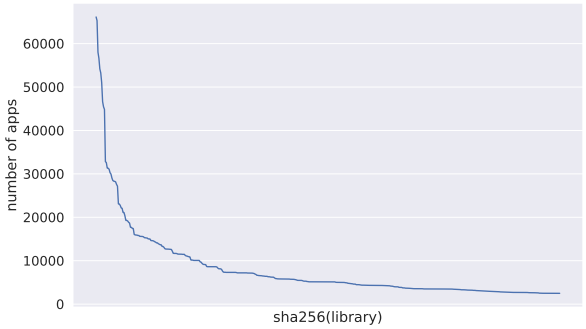


Fig. 2: Number of apps using the 500 most popular native libraries, based on their hash.

Developer	Package name	No. of apps
Facebook	react-native	668,061
Facebook	react-android	334,504
Facebook	folly	53,140
Barteksc	pdfium-android	44,718
Facebook	nativeimagefilters	36,500
Facebook	nativeimagetranscoder	36,149
Getsentry	sentry-android-ndk	35,630
Google	glog	32,444
Google	renderscript	30,703
Google	android-ndk	28,413
Facebook	fbjni	28,024
Brateman	android-sdk	25,049
Facebook	imagepipeline-native	21,406
Facebook	imagepipeline	19,894
Google	flutter	19,157
Facebook	animated-gif	15,460
Yalantis	ucrop	11,541
Facebook	webpsupport	11,501
Ouleiqi	AndroidUtils	7,900
Team-Delta	google-webrtc	7,798
Subodh	conceal	7,111
Facebook	hermes-android	6,818
Koral	android-gif-drawable	6,529
Zetetic Support	android-database-sqlcipher	5,506
Mobius	toutiao	4,822
Yodo1	mmy-core	4,234
Ttvcloudproducts	ttsdk-ttapplog	3,581
Scottyab	rootbeer	3,439
Agorabuilder	full-sdk	3,170
Alviere	alcamera	2,864

TABLE I: Top 30 most popular packages containing native libraries used by Play Store apps

Source	Usage – Global	Usage – Play Store
Jitpack	27,843 (0.39%)	6,277 (0.26%)
Flutter	55,063 (0.78%)	19,157 (0.80%)
React-Glog	84,182 (1.19%)	32,444 (1.35%)
Android-Ndk	99,466 (1.41%)	28,413 (1.18%)
React-Folly	141,503 (2.00%)	53,140 (2.21%)
Renderscript	166,341 (2.35%)	30,703 (1.28%)
Npm	700,343 (9.91%)	328,161 (13.67%)
Maven Central	2,790,966 (39.50%)	1,088,511 (45.36%)
Unknown	2,999,423 (42.45%)	812,985 (33.88%)

TABLE II: Number of native library usages identified from each source.

source, followed by npm thanks to the high number of apps using React Native. Overall, our approach lets us identify 65% of all the library usages on the Play Store.

1) Popular Native Libraries: Studying the popularity of the libraries in our dataset yields some interesting observations. Figure 2 shows how many apps are using the most popular libraries, based on their hash. The plot is clipped to the 500 most popular libraries, as the distribution has a long tail. A close look at the curve in the plot can highlight the presence of flat horizontal components in a plot that is otherwise smooth. This suggests that many apps are using the exact same set of native libraries, tens of thousands of times. The metadata collected from the sources above indicates that the most widely used native libraries overwhelmingly trace back to a small set of popular packages (see Table I).

B. Reliability of Ground Truth (RQ2)

Although we utilize heuristics for frequency-based origin attribution, our ground truth collection relies on the metadata provided by library developers and third-party distribution platforms like Maven. We identify two major potential sources of errors.

1) Human error when inserting package metadata.
2) Inclusion of external binaries in the Maven package without explicitly declaring them as external dependencies in the package metadata.

To determine to what extent those two issues impact our data, we perform a manual evaluation similar to the one performed by Librarian and OSSPolice [1], [2] when evaluating the accuracy of Android native library identification. Namely, we examine apps downloaded from F-Droid [6] as they are open-source and they allow us to easily verify by hand where the libraries are imported from. Inspecting the `build.gradle` file (where dependencies are declared) allows us to verify if the libraries included in the APK come from a specific Maven package or from some other source.

		Potentially Vulnerable Apps by Downloads				
Library	CVE	> 100M	> 10M	> 1M	Total vulnerable	Total apps
libdroidsonroids	CVE-2019-11932	4	35	192	3,411 (58.91%)	5,790
libpdfium	CVE-2018-6120	1	32	314	16,498 (98.86%)	16,688
uCrop	CVE-2018-7641	3	30	217	11,794(100.00%)	11,794
libwebp	CVE-2018-25013	2	39	133	1,104 (97.87%)	1,128
libfolly	CVE-2021-24036	4	16	233	20,386 (65.46%)	31,145

TABLE III: Apps shipping vulnerable libraries by download count.

We increase the scope by including every single native library found on F-Droid apps. We analyze 3,411 F-Droid apps, identify 174 apps bundling native code and extract 1,154 native libraries. After deduplicating binaries with the same hash, we are left with 645 unique binaries with a matching hash in our database. We manually look at each of the 645 matched libraries and check if the metadata for that library is correct (e.g., if the version of the Maven package corresponds to the version of the library).

The manual evaluation reveals that 592 (91.7%) of the libraries are correctly identified (the artifact metadata corresponds to the real developer and version). The errors are relative to packages bundling libraries either copied or compiled from another package without declaring them as explicit dependencies. Most of the 53 errors are caused by the same developer including multiple libraries from different projects in the same package. For instance, there are 20 different versions of `libyoga.so` that are identified as part of React Native. However, we consider those as errors because, although 'yoga' is developed as part of React Native, it is also a standalone component that can be used independently of React Native. A small portion of the errors are more serious. For example, 4 instances of `libgdx.so` are wrongly identified as coming from the package nvlist-android. Further inspection reveals that the libgdx Maven package incorrectly includes the native libraries in the `.jar` java part of the package. Maven requires native libraries to be included in a separate zip archive with the `.aar` extension (cf. Appendix B1).

We conclude that while package metadata cannot always be trusted, our findings indicate that it is accurate most of the time. We consider it sufficiently trustworthy for use in the rest of our analysis.

C. How Outdated are Native Libraries on Android? (RQ3)

Figure 3 shows the average age of libraries used by apps, depending on the number of downloads reported by the Play Store. Figure 3a is particularly interesting as it shows the cumulative distribution of the age of a library.

We can see how, on average, libraries are about 3.5 years old. We also observe that popular libraries are not necessarily more up-to-date than less popular ones. In fact, popular apps with over 10M downloads are slightly slower in updating libraries, with the average library being 3.8 years old.

D. How Many Apps are Potentially Vulnerable? (RQ4)

To estimate the spread of unpatched vulnerabilities in the Android native ecosystem, we adopt a conservative approach and test for vulnerability to a subset of famous, critical, and old CVEs. By focusing on these CVEs (which are the most likely to have been patched) we establish a lower bound on the spread and impact of outdated libraries.

Our selection criteria for these CVEs are the following:

- Large impact: the CVE affects one of the top 30 most popular native libraries (Table I).
- High severity: the CVE can lead to memory corruption and potentially code execution.
- Longstanding issue: the CVE was reported at least 4 years ago.

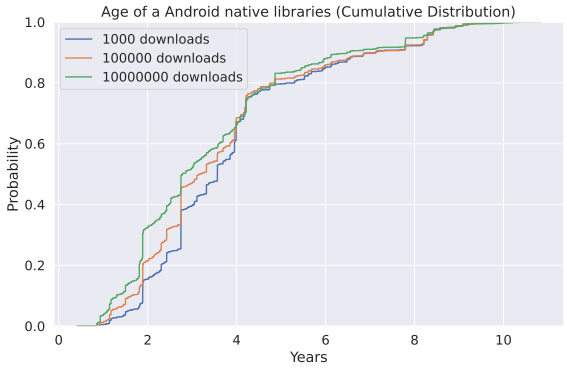
Table III shows a summary of our results. Since these are all well-known CVEs, we expect that some vendors may have backported security patches for them. However, out methodology is unaffected: any such patch would change the `.text` section, producing a different hash that would not match the vulnerable ground-truth entry. Note that for all the CVEs we consider, there is at least one app with over a hundred million downloads that is affected.

It is important to clarify that we conservatively flag apps as *potentially* vulnerable. Exploitability depends on reachability and exposure, which we study in Section IV-F.

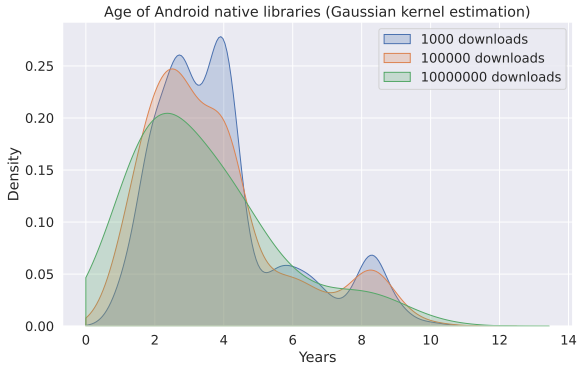
1) Android Gif Drawable (libdroidsonroids): A very popular library used by apps such as WhatsApp, it made the news in 2019 when CVE-2019-11932 was reported. The corresponding bug is a particularly dangerous use-after-free as multiple proof-of-concept exploits are readily available on the web. To determine which apps were vulnerable, we check all apps using the Maven package shipping `libpl_droidsonroids_gif.so` with a version lower than `1.2.18`.

The result is that 80.43% of the 59,660 apps using the library are vulnerable to CVE-2019-11932. If we restrict to apps that are currently available on the Play Store, we find that 58.91% of the 5,790 apps are vulnerable. Vulnerable apps include extremely popular apps such as de.[redacted] with over 400 million downloads.

2) Chromium and AOSP (libpdfium): Since pdfium CVEs do not report the library version but rather the version of Chromium that ships it, we can infer a conservative lower bound on the number of vulnerable apps based on the pub-



(a) Cumulative Distribution of the age of native libraries.



(b) Kernel Density Estimation of the age of native libraries.

Fig. 3: Visualizations of the age of native libraries used by Android apps available on the Play Store.

lished date of the pdfium-android package. For example, for CVE-2018-6120, we consider all apps using a version of `libpdfiumjni.so` published before April 17, 2018 (the date the vulnerability was disclosed to Google), as vulnerable.

In total, 99.34% of the 79,414 affected applications are vulnerable to CVE-2018-6120. Play Store apps show a similar number, with 98.86% of the 16,688 apps marked as vulnerable. This high percentage arises because every version of the pdfium-android package bundles the same outdated copy of `libpdfiumjni.so`, released seven years ago, thereby making any app that depends on the package vulnerable.

3) uCrop (CImg): uCrop is a particularly telling case. The library performs cropping operations on a variety of media formats. Dependencies such as `CImg`, `libjpeg.so` and `libpng.so` are statically linked into the library, but the version is clearly denoted in the upstream repository.

Unfortunately, while uCrop is still actively developed, the bundled version of `CImg` currently included is from 10 years ago, which is vulnerable to at least CVE-2018-7641. An issue pointing this out was opened in 2022 in the uCrop GitHub repository. However, one of the contributors responded with the following:

> “As far as I remember (complied [sic] these native libs 6 years ago) - all the 3rd-party libraries like libpng & libjpeg were static global dependencies (basically, C++ files on my laptop) that were compiled into the .so files [...]
> In order to recompile the files inside the jniLibs folder, one should have the latest libpng & libjpeg dependencies available on their development machine, updated Android.mk & Application.mk scripts, some luck & use ndk build.
> Easier said than done, but that’s how the opens source [sic] works.”
>
> *2023 — https://github.com/Yalantis/uCrop/issues/843*

As a consequence, 100% of the 32,810 apps using uCrop (11,794 currently on the Play Store) are vulnerable to CVE-2018-7641. More generally, they are potentially vulnerable to any libPNG or libJpeg CVE reported after 2017.

```
;function WebPGetDecoderVersion:
mov w0, 2 ; store 2 in the first byte
movk w0, 1, lsl 16 ; store 1 in the third byte
ret ; return value 0x00010002 (i.e. 1.0.2)
```

Listing 1: ARM64 disassembly of the function WebPGetDecoderVersion for libwebp 1.0.2

4) Imagepipeline (libwebp): Facebook’s imagepipeline (also known as “Fresco”) is a powerful framework to display images on Android. One of their dependencies is `libwebp.so`, a WebP parsing library developed by the WebM group. In 2018 the oss-fuzz project [7] found a number of heap buffer overflows in libwebp, identified in the CVE range from CVE-2018-25009 to CVE-2018-25013. These affect libwebp versions before `1.0.1`.

However, identifying the version of the bundled `libwebp.so` is not trivial. Fresco does not declare it as a specific dependency, but instead downloads and compiles it on the fly as part of their build process. To address this, we exploit the fact that libwebp exposes a function called `WebPGetDecoderVersion` that returns an integer encoding the major, minor, and micro version (according to semantic versioning [8]) in the first three bytes of the integer. Listing 1 shows the disassembly of the function. Since it is simple and short, we can emulate the function using radare2’s [9] ESIL emulation [10] to retrieve the value of register `x0`. Listing 2 shows the Python code to emulate the function and parse the result.

In total, 99.60% of the 20,392 apps use a bundled `libwebp.so` version before `1.0.1`, vulnerable to the previously mentioned heap overflows. On the Play Store, 97.87% of the 1,128 apps using libwebp are vulnerable.

5) React Native (libfolly): Folly is Facebook’s extension to the C++17 standard library. It is very popular among Android apps as certain modules of React Native make extensive use

```python
from packaging.version import Version
cmd = "s sym.WebPGetDecoderVersion;"
cmd += "aei;" # reset ESIL state
cmd += "aeim;" # reset ESIL memory
cmd += "aes 3;" # emulate 3 instructions
cmd += "aer x0;" # print ESIL register 0
version = subprocess.check_output(
   f"r2 -qc '{cmd}' {file}", shell=True)
major = (int(version, 16) >> 16) & 0xff
minor = (int(version, 16) >> 8) & 0xff
micro = int(version, 16) & 0xff
version = Version( f"{major}.{minor}.{micro}")
if version < Version("1.0.1"): print(f"vulnerable")
```

Listing 2: Python code to verify if the version of a libwebp library is below 1.0.1

of `libfolly-runtime.so`. CVE-2021-24036 is a heap buffer overflow patched in version `v2021.07.22.00`. In total, 82.39% of 81,920 apps using Folly are vulnerable to CVE-2021-24036. On the Play Store, 65.46% of the 31,145 apps are vulnerable.

6) Summary of Results and Potential Bias: Out of the 66,545 Play Store apps bundling one of the five considered libraries, 79.9% still ship a vulnerable version.

Our analysis is limited to five CVEs, which can bias the absolute prevalence and should not be interpreted as a claim about *all* Android native-library CVEs. However, we intentionally focus on old, high-severity, plausibly triggerable memory-corruption vulnerabilities—cases that should be *most likely* to have been patched after years of exposure—so the observed 79.9% unpatched rate should be read as a conservative lower bound on the ecosystem's remediation performance. The broader library-age trends (Figure 3) and our independent check on widely used libraries outside the ground truth (Section IV-E) corroborate that the issue is not specific to these five case studies.

E. Libraries Outside Ground Truth (RQ5)

Our ground-truth data is unfortunately incomplete due to libraries such as `libpng.so` often being statically compiled. In fact, many traditional system libraries do not have an official package in any of the sources we analyzed. For this reason, developers often statically link them in their apps. Our approach can be extended to support such cases (e.g., see uCrop in Section IV-D3) However, this requires manual effort and cannot scale to the entire dataset.

To estimate how outdated libraries outside our ground truth are, we use a lightweight heuristic. We rely on Librarian's approach [1] of extracting library versions with regular expressions. While Librarian uses regular expressions as a primary identification mechanism, we apply them here only as a supplementary heuristic on four libraries with well-defined version string formats, and solely to estimate distributional trends.

We apply this approach to four widely used libraries that lack packaged artifacts in our ground-truth sources: FFmpeg, libpng, OpenCV, and libjpeg-turbo.

We determine the release date for each version by examining the timestamp of the git commit associated with the version's release, found in the libraries' repositories. For each of those libraries, we also select the latest high-severity (memory corruption) CVE reported and calculate how many apps are shipping vulnerable versions of the library.

We plot the number of apps using libraries released in the last 10 years. To compare against ground-truth data, we also include in the plot Android-Gif-Drawable (`libpl_droidsonroids_gif.so`) which comes from Maven Central. The results are shown in Figure 4. In the plot, we also highlight the notification date of the high severity CVEs for each of those libraries. All the apps using libraries released before the notification date of the CVEs are considered vulnerable and highlighted in red in the figure.

There are two interesting observations from the plot. First, we can see that the number of vulnerable apps is very high for all libraries. libjpeg-turbo is particularly problematic as 99.7% of the apps using it are vulnerable to CVE-2023-2804. FFmpeg is also showing a high percentage of vulnerable apps, with 82.8% of the apps marked as vulnerable to CVE-2020-12284. Second, we note how the cumulative distributions of the age of the libraries (marked with a blue line in the plot) follow a similar pattern to the one of Android Gif Drawable. This suggests that the percentage of apps shipping outdated libraries is similar, and that the issue we are observing is not limited to libraries in our ground-truth data, but it is present in the whole Android ecosystem.

F. Real-World Impact (RQ6)

For an app to be at practical risk, it must not only include a vulnerable library but also expose the affected code paths to attacker-controlled input. If the vulnerable functions are never invoked or inputs are sufficiently constrained, the vulnerability can remain unreachable in typical use.

We adopt a stricter Android threat model than the one used in prior work: the attacker can deliver untrusted input to an application (e.g., via network content, Intents, or local files) but its goal is not just leaking private data but rather triggering memory corruption in the context of the app. While modern Android OS mitigations (e.g., the Scudo hardened allocator) make achieving full arbitrary code execution challenging and often fail-safe into an application crash, proving reachability of memory corruption primitives still highlights a severe risk. At a minimum, this constitutes a remote Denial of Service (DoS), with potential for code execution on older devices lacking such heap hardening.

Our methodology for assessing whether an app is exploitable is as follows:

- Reverse engineer the app to identify library call sites.
- Determine if the data passed to the native function is attacker-controlled.

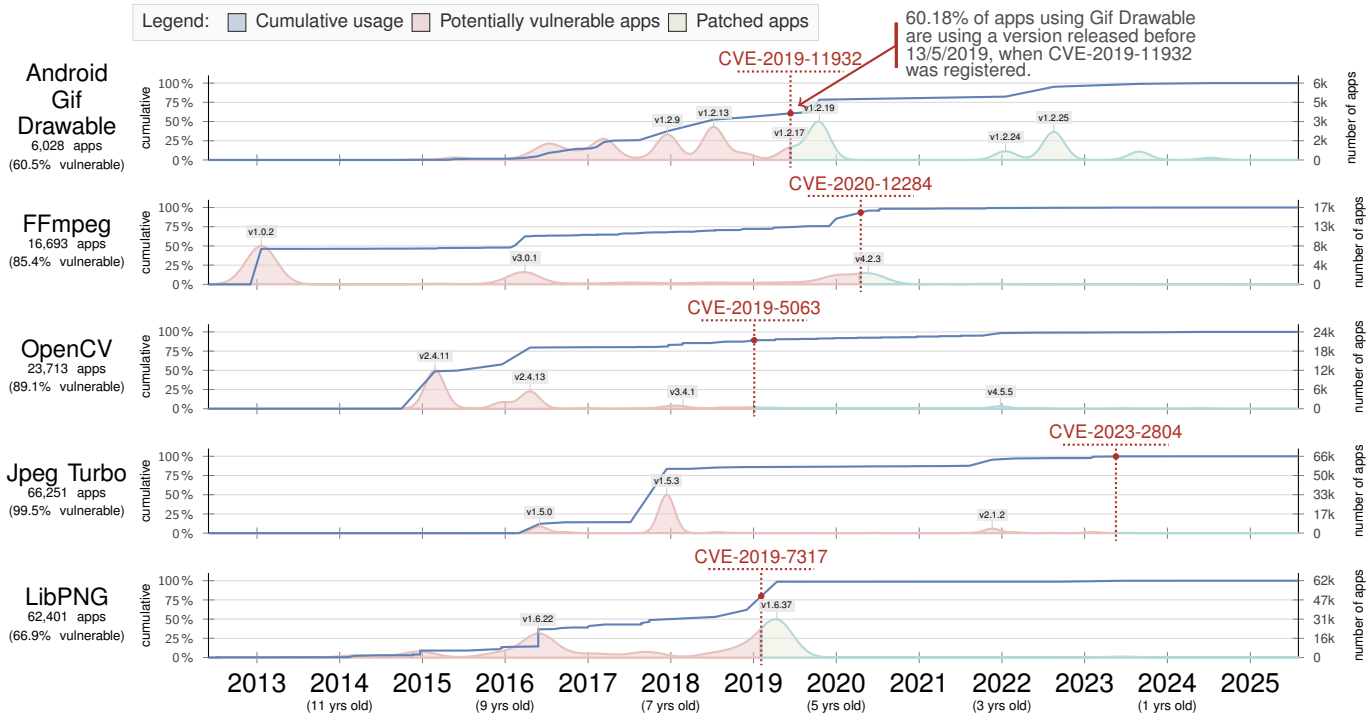


Fig. 4: Age of popular native libraries found in current Android apps on the Play Store. Apps that use library versions older than the sample CVEs, may be vulnerable to attacks. The blue line shows the cumulative total of apps (where this line crosses a CVE it highlights the percentage of vulnerable apps). The red and green areas highlight the distribution of apps that use a vulnerable/updated version of the library.

- Install the app on a test device and manually interact with the app, relying on Frida hooks to determine if the native function is called.
- Emulate an attacker by delivering the malicious input through the app's exposed vectors (e.g., network interception, file sharing).
- Confirm the memory corruption is reached via logcat to check if the app crashed inside of the vulnerable native library due to the vulnerability.

Section IV-F4 provides a step-by-step walkthrough of the methodology on a case study. Table IV summarizes the results.

To assess and estimate real-world impact, we manually analyzed the 100 most popular apps affected by three libraries: libpdfium, libdroidsonroids, and libucrop. We selected these libraries because they all parse media often supplied by users.

A large chunk of the apps we considered require an authenticated account to use them, or include anti-debugging techniques. We exclude those apps from our analysis. Apps that we are able to successfully analyze are marked as "Testable" in Table IV. From an initial pool of 300 apps, we managed to test 160.

1) libpdfium: CVE-2018-6120 is a heap out-of-bounds write during PDF rendering. Among 100 apps bundling a vulnerable version, 49 are "testable" (no anti-debugging, no login required).

We are able to call the vulnerable function on 37 apps. Out of those, 15 apps parse PDFs from an untrusted source. We are able to trigger the crash on all 15 apps. Notably, three of those 15 apps are banking apps.

	pdfium	roidsonroids	ucrop
Number of apps	100	100	100
# Testable apps	49	68	43
# Vulnerability reachable	37	47	20
# Attacker-controlled input	15	10	20
# Crash triggered	**15**	**7**	**12**

TABLE IV: Results of the exploitability analysis.

2) libdroidsonroids: CVE-2019-11932 is a double-free vulnerability that can be triggered when rendering malformed gifs. Among the 100 most popular apps bundling a vulnerable libdroidsonroids, we verify that 47 actually invoke it to render gifs. Of these, 10 display GIFs from external sources. We trigger the crash in 7 of those 10 apps.

Of the seven exploitable apps, one is a gallery app that loads gifs from the sdcard. The remaining six download gifs over the network from untrusted sources (one retrieves gifs from `giphy.com`, one from social media posts, and four via a third-party advertising provider).

3) libucrop: uCrop's exploitability is more nuanced. uCrop bundles three 10-year-old native components (libpng, libjpeg,

```
signal 11 (SIGSEGV), code 2 (SEGV_ACCERR)
  ...
  backtrace:
  #00 libc.so (memset+88)
  #01 libpl_droidsonroids_gif.so
  #02 libpl_droidsonroids_gif.so
  #03 libpl_droidsonroids_gif.so (bindSurface+1236)
  ...
```

Listing 3: The logcat crashlog.

and CImg) affected by multiple CVEs and used in the cropping pipeline. By design, uCrop first decodes and renders the image using Android's built-in Skia library. Only after the user clicks on the crop button, does libucrop generate the cropped image. If Skia fails to decode the input, uCrop returns early and the native cropping routine does not run.

This initial Skia decode effectively acts as input validation and blocks many crashing proof-of-concept inputs for known libpng/libjpeg CVEs. However, we find that certain .bmp images triggering CVE-2018-7641 (heap buffer overflow) decode successfully in Skia and then crash inside libucrop.

Out of the 100 apps we inspect, 20 use uCrop to modify images loaded from shared storage. However, in 8 apps the exploit fails during the Skia decoding phase. In total, we trigger a crash in libucrop for 12 apps.

4) Case Study: Org.[redacted] is a social media client for Twitter/Mastodon with around half a million downloads. The app uses a vulnerable version of libdroidsonroids. Reverse engineering the app we find that a malicious Mastodon instance could serve a GIF that triggers CVE-2019-11932.

We identify the following data flow:

- The MediaDownloader class abstracts the media downloading functionality of the app.
- Depending on the social media platform, it uses different methods to extract media URLs (e.g., TwitterMediaProvider).
- A ParcelableMedia object is instantiated with the media URL, and sent to the MediaViewerActivity.
- Depending on the media type (image, video, or GIF), a specific viewer is chosen to display the media.
- If the URL ends with .gif, the GifPageFragment is used to display the GIF. The raw bytes of the GIF are transformed into an InputStream by a call to InputSource.UriSource(context.getContentResolver(), uri).
- The InputStream is then passed to the native library with a call to GifInfoHandle.openStream(InputStream).

This confirms that attacker-controlled input reaches the vulnerable function. We then install the app and connect it to our own Mastodon instance. We serve a GIF crafted to trigger the double-free. Listing 3 shows the relevant logcat excerpt: the app crashes while processing the GIF inside the native library, confirming the vulnerability.

G. Case study: Automating Reachability Analysis with AI Models

While our manual analysis confirms the presence of real-world attack paths, it is particularly labor-intensive as it requires reverse engineering and manual interaction with each app. We run a small case study to determine if state-of-the-art AI models could potentially automate the reverse engineering part of the process, which is the most time-consuming part. We use the same methodology as before, but instead of manually analyzing the app, we prompt Gemini 3.1 Pro with the decompiled code and ask to verify if the app is vulnerable and to provide a step-by-step analysis of how the data flows from any untrusted input to the vulnerable function.

To determine the accuracy of the model, we run the experiment on the same subset of apps we analyzed manually and compare the results. We allocated a total budget of 300 USD for the experiment, which allowed us to analyze 78 apps. When comparing the results, we find that the model is able to correctly mark 72 out of the 78 apps as vulnerable or not, justifying its answer through the data flow analysis. Notably, the 6 mistakes are all false positives, where the model marks the app as vulnerable but it is not. After manually inspecting the errors, we find that they are all caused by the model being unable to identify if a given network resource is downloaded through a secure channel (e.g., HTTPS) or not.

While the number of apps tested is too low to draw any conclusions, the results suggest that AI models show promise in automating the reverse engineering part of the process, and that they are able to navigate the complexity of real-world Android apps. We suspect a more complete harness that also allows to automate the interaction with the app (e.g., using a tool such as DroidBot [11]) could potentially extend the automation to the whole process, including the triggering of the vulnerability.

H. Discussion of the Results

a) Real-world impact: Even under conservative assumptions that likely understate the true impact, our exploitability study (Section IV-F) shows that the risk to end users is real and substantial. Across a stratified sample of 160 Play Store apps bundling vulnerable versions of three widely deployed native libraries (libpdfium, libdroidsonroids, and libucrop), we reliably triggered memory corruption crashes in 34 apps (see Table IV). These include high-profile apps—such as banking apps and cryptocurrency wallets—still available on the Play Store. In many cases the attack surface is reached through routine interactions (e.g., opening a PDF, rendering a GIF from a remote feed or advertisement SDK). Even when front-end decoders (e.g., Skia) block malformed inputs, we observed TOCTOU windows and shared-storage reload patterns that bypass these gates in real apps. Because around half of the candidates could not be tested due to strict authentication and anti-debugging, our 34/300 (11%) figure is a lower bound on this sample. It is likely that the true figure is higher, since we are assuming that none of the 140 apps we could not analyze are vulnerable. The takeaway is that outdated native

libraries yield reachable, user-facing remote exploitation paths that reliably cause Denial of Service and potentially lead to code execution on unhardened devices.

b) A systemic issue: The point of this study is not to identify vulnerabilities in specific apps, but rather to uncover a general trend in the Android ecosystem. In Section IV-D, we rely on our ground truth and consider only old CVEs, and for all of them we find an alarming amount of vulnerable apps. The experiment in Section IV-E demonstrates that the trend is observable independently of our specific ground truth set. In fact, using version strings to estimate the age of common statically compiled libraries yields very similar results to the analysis from Section IV-C. We conclude that outdated native libraries are a widespread issue in the Android supply-chain ecosystem. Only a small minority of developers keep up with the latest security patches. Hence, any kind of solution must target the whole ecosystem, not just a few specific libraries.

c) Responsible Disclosure: Consistent with responsible disclosure practices for large-scale ecosystem vulnerabilities, contacting every potentially affected developer individually is infeasible given the sheer volume of applications identified. We are coordinating our disclosure efforts with the Google Android Security team. We securely shared our detailed findings to enable Google to utilize their established channels (e.g., Google Play Console warnings) for notifying developers.

I. Comparison with Prior Work

1) Librarian: Librarian's approach relies on two primary mechanisms for library version identification: (1) computing a Jaccard-based similarity score using five metadata features extracted from ELF binaries (exported/imported functions, exported/imported globals, and library dependencies), and (2) applying regular expressions to strings in the .rodata section to extract version information. Its design exhibits several fundamental limitations that constrain both accuracy and scalability. We compare our approach to Librarian on two fronts: scalability and accuracy.

a) Scalability: Librarian's approach requires manual labeling of "good" libraries to build a ground-truth dataset from which features are extracted. This manual curation process is labor-intensive. In fact, Librarian's evaluation used only 46 distinct libraries with 904 versions. Furthermore, the dataset of labeled libraries requires continuous maintenance to keep up with new library releases. Additionally, the regular expressions for version string extraction are also hand-crafted for each library, requiring human involvement in this step as well.

These manual steps severely limit Librarian's scalability. In fact, the authors analyze only 904 libraries as their ground truth. In contrast, our approach is fully automatic and scales to millions of libraries and apps.

b) Accuracy: We are unable to evaluate the Jaccard similarity component of Librarian, as the authors do not provide the code to build a new dataset of labeled libraries (the prebuilt dataset included with the paper from 2021 is too outdated, focusing on arm32 libraries). However, we can directly assess the accuracy of their version string extraction mechanism. Librarian relies on version strings when the Jaccard similarity is inconclusive. On their own evaluation dataset, the authors report that this is the case for about half of the binaries. If we assume this to be true for our dataset as well, we can still infer a reasonable estimation of the accuracy of the overall approach of Librarian by just looking at the version string extraction accuracy.

Librarian's reliance on regular expressions for version string extraction is inherently brittle and prone to error. For example, consider the regular expression used to identify sqlite3:

```
^3\.([0-9]{1,}\.)+[0-9]}
```

It can match unrelated strings like `libcurl/3.1.4` and only partially match valid tags such as `3.45.2-rc1`.

Another example is the regex for identifying Unity3D:

```
([0-9]+\.)+([0-9]+)[a-z][0-9]|Expected version:(.*)
```

This regex has two branches separated by the pipe symbol. The `Expected version:(.*)` branch could match arbitrary log text, and the numeric branch matches any generic version substring (e.g., `1.2.3a4`).

We can assess how often such inaccuracies occur in practice by running Librarian's version string extraction on our F-Droid dataset (Section IV-B), where we have ground-truth library versions for 592 unique binaries. In this dataset, there are 18 sqlite3 binaries and zero Unity3D binaries. However, Librarian's regexes perform poorly: it identifies 36 binaries as sqlite3 (only 1 of them is correct, the other 35 are false positives). Unity3D is even worse: the regular expression identifies 57 binaries as Unity3D, but they are all false positives. These results are very far from the overall 91% accuracy reported by the authors [1]. We hypothesize this discrepancy is due to Librarian's evaluation being performed on the same dataset used to craft the regular expressions.

Due to limitations with the Librarian open source release, we can only compare against one component. Since it constitutes roughly half of their matches, our analysis still demonstrates the benefits of ground-truth matching compared to string-based heuristics.

2) OSSPolice: OSSPolice [2] builds an index by comparing binaries against source code repositories, measuring similarity between strings extracted from binaries and the ones found directly in source code. The main limitation of OSSPolice's approach is its reliance on source code availability, excluding from the analysis any closed-source libraries. Furthermore, OSSPolice's string-based similarity approach is also prone to inaccuracy as many open-source projects include third-party libraries in their source code, with the potential to generate false positives.

Unfortunately, OSSPolice cannot be directly compared against. The Librarian authors report that despite repeated attempts to obtain a working version of OSSPolice from its creators in 2021, the authors did not provide their non-public dataset. OSSPolice reports a small evaluation on 67 native libraries pulled from F-Droid in 2016. The tool achieved 82% accuracy. On the same dataset, Librarian reports a 94%

accuracy. It is unfeasible to test our approach on the same 67 libraries as they are either not available anymore or are 32-bit only, which we do not support (as systems have moved to 64-bit).

J. Potential Causes

The extremely high percentage of vulnerable apps points to a community-wide neglect of updating native packages. The most likely explanation is that the average app developer downloads the required native libraries once and then never updates them again. The issue is so prevalent that it suggests the problem is not in the individual developers' behavior, but rather in the whole supply-chain process of Android libraries. We now point out what we believe are the two root causes that led to this situation:

a) Lack of incentives from the tooling: Even when native libraries are correctly specified as Maven dependencies in the `build.gradle` file, Gradle will not automatically check for updates. To force a re-download of the latest libraries, developers need to manually run `gradlew build --refresh-dependencies`. There is a Gradle plugin called DependencyCheck [12] that scans the National Vulnerability Database (NVD) for relevant CVEs for dependencies declared in the `build.gradle`. However, it requires manual setup, plus requesting an NVD API key. Android Studio automatically checks for dependency upgrades, and inserts linter hints in the `build.gradle` file where outdated dependencies are specified. Outdated dependencies are also listed in the "Suggestions" window of Android Studio, but the user needs to manually open it from the "File > Project Structure" menu. Furthermore, many app developers do not rely on Android Studio (especially for large apps), using Gradle directly or relying on another build system wrapping Gradle.

b) Lack of incentives by the ecosystem: In the past, Google took steps to address vulnerable software in Play Store apps. Notably, in 2016, Google reached out [13], [14] to developers bundling a vulnerable libpng in their app to protect users against CVE-2015-8540. The company informed developers that new updates or releases of apps with this vulnerable version of libpng would not be allowed on the Play Store. This initiative was part of a campaign called the ASIP [15] (App Security Improvement Program). ASIP focused on scanning Play Store apps for common vulnerabilities (e.g., SQL injection, ZIP file path traversal). An app flagged by a vulnerability scan would either receive a warning or a "remediation deadline". After the deadline, apps still vulnerable faced potential removal from the Play Store [16]. Some of these scans were relative to native libraries. In 2015, GnuTLS was part of a campaign. In 2016 libupnp, libpng, libjpeg-turbo, and openssl were targeted. To the best of our knowledge, no scans targeted vulnerable libraries after 2016.

Such an approach becomes infeasible for native libraries bundled in packages where even the newest package version is vulnerable. In such a scenario an app developer wishing to address the vulnerable native library has two options: either forking the package and rebuilding the native library with a newer version or replacing the package with an equivalent one. Both options are extremely costly. There is also no clear incentive for package maintainers to update native libraries, especially if doing so introduces breaking changes or requires significant engineering effort (see Section IV-D3).

V. Proposed Countermeasures

We argue that a threefold approach is required for an effective mitigation of such a pervasive issue:

a) Changes in the Android ecosystem: Updating dependencies requires time and energy, and developers need to have strong incentives to do so. Changing the documentation to strongly highlight the importance of up-to-date libraries is a first step but likely insufficient to change the habits of developers.

Since the real risk is for the users of the apps, we propose that the Play Store should notify users about the potential dangers if they are installing an app with known vulnerable dependencies. Introducing a confirmation prompt that requires user acknowledgment (similar to the one used to sideload APK files) could significantly increase awareness of this issue. This approach serves two primary purposes: first, it informs users about the possible risks, giving them a chance to reconsider installing the app. More importantly, it would act as a strong incentive for developers to quickly fix the issues, since a conspicuous warning might deter users from installing the app.

b) Changes in the Android tooling: We propose to make Gradle automatically scan for dependency updates, and alert developers of potential security updates. However, it is important to prevent scenarios similar to npm audit, where automatic checks lead to low signal-to-noise ratio driving developers to ignore those alerts. We suggest that during normal development, Gradle reports outdated dependencies in the logs. However, the behavior is different when building with the "release" target. If potential known security issues are found in the outdated dependencies, Gradle would then require the developer to manually confirm and acknowledge that they are building an app shipping vulnerable code.

c) Stronger operating system mitigations: Android already has existing mechanisms to isolate apps from each other. An exploited native library can grant the attacker access to an individual app's data. We believe that the Android ecosystem should take a more in-depth sandboxing approach. For example, native libraries themselves could be run in an isolated environment, applying the principle of least privilege and granting the native code only access to the strictly necessary resources. Additionally, Android should adopt modern binary mitigations like the ones employed by GrapheneOS [17]. These include a secure application spawning system that enables Address Space Layout Randomization (ASLR), a hardened memory allocator, Shadow Call Stack (SCS), Branch Target Identification (BTI), Pointer Authentication Codes (PAC), Memory Tagging Extension (MTE), disabled dynamic code loading, and more.

VI. Threats to Validity

While our analysis is based on ground-truth data, there are a number of cases that can lead to mistakes in our analysis.

a) Incorrect Maven metadata: The metadata found on each Maven package was taken at face value. There could be mistakes in the information about the developer or version of the library. In Section IV-B we manually verify a sample of the metadata against apps from F-Droid. We find that 592 out of the 645 (91.7%) libraries are correctly identified (the artifact metadata corresponds to the real developer and version). A potential limitation of this verification is that F-droid apps might not be representative of the broader Play Store population, as they are open-source, generally smaller and they lack frameworks such as advertising SDKs.

b) Incorrect library names on Maven: In some cases we take the library name into account as part of the process described in Section III-B. However, library names can be arbitrarily changed by developers. To understand how often developers rename libraries during repackaging, we conduct a small study on Play Store apps. We analyze all the 5,790 apps that use the Android Gif Drawable Maven package, and find that only 11 of them had renamed the library in the APK.

c) Static compilation: Libraries shipped via Maven packages might statically include other libraries that would elude our analysis. This phenomenon is unfortunately quite common. We analyze the OpenSSL library as an example to get an idea. By extracting version strings like suggested by previous work [1], we obtain a rough estimate of how many binaries statically link against openssl. We used an updated regex from the one used in Librarian [1]. We found a total of 2983 binaries matching our regex. However, only 64 of those binaries have names suggesting an SSL library (i.e., `libcrypto.so`, `libssl.so`, `libopenssl.so`, etc). More worryingly, out of the 228 Maven artifacts shipping those binaries, only one was actually called "openssl". The vast majority of the others were projects that directly compiled OpenSSL into their native library, or that included libraries statically linked against OpenSSL.

d) Scope limited to aarch64: Our study only takes into account aarch64 libraries. Since the Cortex-A715 and Cortex-A520 models introduced in 2022 and 2023 respectively, ARM stopped the production of 32-bit mobile processors [18]. Therefore, we exclude arm32 libraries from our analysis.

e) Libraries loaded at runtime: Our approach extracts libraries from the `/lib/arm64-v8a/` folder, as they are automatically parsed and loaded by the DVM at app start. While the vast majority of applications follow this convention, apps may store libraries in different folders (e.g., in the resources folder) and manually load them via APIs such as `System.loadLibrary`. To estimate the impact of this issue, we perform a small study on 1,000 apps that used native libraries, looking for shared object files outside the folder `/lib/arm64-v8a/`. We did not find an app that contained native aarch64 libraries outside the `/lib/arm64-v8a/` folder.

f) Selection bias in libraries coming from popular sources: Our study does not consider libraries for which we could not track the origin of. This includes open-source libraries custom patched by developers, or in some cases libraries compiled with a different compiler version than the one used by the original developer. Considering only libraries that we download from popular sources (Maven, Jitpack, etc.) may skew the results. While we cannot address this issue entirely, we believe its impact is limited. In fact, libraries from popular sources consist of about 65% of the overall native library usages in Android apps (Figure 1), which is a large enough portion to be representative of the ecosystem.

VII. Related Work

a) Library identification.: Librarian [1] is the first work that called attention to how outdated native libraries on Android really are. The authors identified library versions using a Jaccard similarity score, using binary symbols as features. Additionally, they relied on regex pattern matching to extract version strings for a few dozen common libraries. The study was limited to the 200 most popular apps; however, the reliance on heuristics limited the accuracy. Nonetheless, the authors correctly identified a number of vulnerable and outdated libraries, claiming that 26% of the apps were using libraries with known vulnerabilities. A more detailed comparison with Librarian is in Section IV-I.

OSSPolice [2] conducted a large-scale study of over a million apps trying to identify libraries in Android apps (both Java and native ones). They extract features such as strings from binaries and compare them to an index built by scraping Maven, GitHub and JCenter for popular open-source projects. Since the detection is based on heuristic similarity matching [19], their reported accuracy on a small dataset of 67 libraries is limited and lower than Librarian [1].

b) Android native libraries.: Many approaches have been proposed to detect vulnerabilities in Android native code. Park et al. [20] relied on decompilation to then apply static analysis techniques on source code. A more recent study from 2024 worked on assigning a risk "score" to libraries [21]. ATLAS [22] and Droidot [23] use fuzzing to uncover bugs in Android native libraries. DroidReach [24] aims to determine the reachability of native functions from attacker-controlled input, relying on Angr [25] and Ghidra [26] to perform static analysis. On a similar note, Ndroid [27] explores Java-native interfaces for information leaks, using QEMU.

A study from 2014 [28] was the first to crawl the Play Store to categorize apps. Another large-scale study was conducted in 2016 by Afonso et al. [29] over a million Android apps, focusing on collecting various metrics on native code usage to estimate the impact of native sandboxing techniques.

c) Supply-chain attacks.: Package managers and dependency distribution mechanisms have been analyzed before by researchers. He et al. [30] showed how in the popular JavaScript package manager npm version pinning might be

[1] `openssl[ -]\d(\.\d){2}([a-z]|-alpha\d{1,2}|-beta\d|-pre\d)?`

not enough to defend against supply-chain attacks. In 2020, another study [31] identified 339 malicious packages in PyPI, npm, and RubyGems, including some with more than 100,000 downloads. Open-source software is particularly vulnerable to supply-chain attacks, as highlighted by Ladisa et al. [32]. Recent research also focused on the containerization of npm dependencies [33], proposing to apply the principle of least privilege to downloaded packages.

VIII. Conclusion

Four years ago, previous work studied the presence of risky outdated libraries in a subset of Android apps. Our study, which encompasses the entire Android ecosystem, reveals that the extent and severity of the issue were significantly underestimated. Moreover, our findings indicate that there has been no improvement: native libraries are as outdated as ever. Based on our ground-truth dataset, we accurately identify the version of most native libraries in Android apps. We picked 5 critical, longstanding CVEs from four years ago and found they impact at least 53,193 apps currently available on the Play Store that ship vulnerable libraries. Our heuristic analysis on version strings suggests that the issue is not limited to the CVEs we selected.

We believe the problem is endemic, rooted in the ecosystem, and we call for immediate action to address it. We shared our results with Google and proposed mitigations to protect users and help developers automatically update dependencies.

Acknowledgments

We would like to acknowledge the support of the AndroZoo project, which provided the data for this study. Furthermore, this paper was edited for grammar using Gemini 3.1 Pro. This work was supported, in part, by SNSF 10004669 "Lin-Specteur" and SNSF Eccellenza PCEGP2 186974.

Ethical Considerations

Our research involved analyzing a large number of Android applications and their native libraries, which raises several ethical considerations that we carefully addressed.

We have responsibly disclosed our findings to Google's Android Security Team, including detailed information about vulnerable applications and outdated libraries. We are actively working with Google to determine the most appropriate way to notify affected developers without creating additional security risks.

All applications analyzed were obtained through legitimate channels (AndroZoo repository and official package distribution services). We only collected technical metadata and library information, avoiding any user-specific or sensitive data. We are publishing our data with the permission of the AndroZoo team. In our released data, application names are obfuscated.

References

[1] S. Almanee, A. Ünal, M. Payer, and J. Garcia, "Too quiet in the library: An empirical study of security updates in android apps' native code," in *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion*, 2021.

[2] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," in *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*, 2017, pp. 2169–2185.

[3] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2099–2116.

[4] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 468–471.

[5] JoMingyu, "Google play scraper pip package," https://github.com/JoMingyu/google-play-scraper, accessed: 24-03-2025.

[6] "F-droid," https://f-droid.org/en/about/, accessed: 24-03-2025.

[7] M. Aizatsky, K. Serebryany, O. Chang, A. Arya, and M. Whittaker, "Announcing oss-fuzz: Continuous fuzzing for open source software," https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html, accessed: 30-01-23.

[8] "Semantic versioning," https://semver.org/, accessed: 24-03-2025.

[9] "Radare2 reverse engineering framework," https://rada.re/n/, accessed: 24-03-2025.

[10] "Radare2's introduction to esil," https://book.rada.re/emulation/esil.html, accessed: 24-03-2025.

[11] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th international conference on software engineering companion (ICSE-C)*. IEEE, 2017, pp. 23–26.

[12] "Dependencycheck gradle plugin," https://plugins.gradle.org/plugin/org.owasp.dependencycheck, accessed: 24-03-2025.

[13] "Google play security warning," https://github.com/Esri/arcgis-runtime-samples-android/issues/172, accessed: 24-03-2025.

[14] "Message from google to play store developers," https://community.esri.com/t5/arcgis-appstudio-questions/google-play-security-warning/td-p/853541, accessed: 24-03-2025.

[15] "App security improvement program," https://developer.android.com/privacy-and-security/googleplay-asi, accessed: 24-03-2025.

[16] "Libpng asip page," https://support.google.com/faqs/answer/7011127, accessed: 24-03-2025.

[17] "Grapheneos," https://grapheneos.org/features, accessed: 24-03-2025.

[18] "Arm official documentation," https://developer.arm.com/Processors/, accessed: 24-03-2025.

[19] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, 2002, pp. 380–388.

[20] J. Park, S. Lee, J. Hong, and S. Ryu, "Static analysis of jni programs via binary decompilation," pp. 3089–3105, 2023.

[21] S. L. Sanna, D. Soi, D. Maiorca, G. Fumera, and G. Giacinto, "A risk estimation study of native code vulnerabilities in android applications," *Journal of Cybersecurity*, vol. 10, no. 1, p. tyae015, 2024.

[22] H. Xiong, Q. Dai, R. Chang, M. Qiu, R. Wang, W. Shen, and Y. Zhou, "Atlas: Automating cross-language fuzzing on android closed-source libraries," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 350–362.

[23] L. Di Bartolomeo, P. Mao, Y.-J. Tung, J. Ayala, S. Doria, P. Celada, M. Busch, J. Garcia, E. Losiouk, and M. Payer, "Hercules droidot and the murder on the jni express," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025.

[24] L. Borzacchiello, E. Coppa, D. Maiorca, A. Columbu, C. Demetrescu, and G. Giacinto, "Reach me if you can: On native vulnerability reachability in android apps," in *Computer Security–ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part III*. Springer, 2022, pp. 701–722.

[25] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 8–9.

[26] N. S. Agency, "Ghidra software reverse engineering suite," https://ghidra-sre.org/, accessed: 30-01-23.
[27] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. Chan, "NDroid: Toward tracking information flows across multiple android contexts," vol. 14, no. 3, pp. 814–828. [Online]. Available: https://ieeexplore.ieee.org/document/8443386/
[28] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *The 2014 ACM international conference on Measurement and modeling of computer systems*, 2014, pp. 221–233.
[29] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, G. Vigna *et al.*, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in *The Network and Distributed System Security Symposium 2016*, 2016, pp. 1–15.
[30] H. He, B. Vasilescu, and C. Kästner, "Pinning is futile: You need more than local dependency versioning to defend against supply chain attacks," 2025.
[31] R. Duan, O. Alrawi, R. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," 2020.
[32] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," pp. 1509–1526, 2022.
[33] G. Ferreira, L. Jia, J. Sunshine, and C. Kästner, "Containing malicious package updates in npm with a lightweight permission system," pp. 1334–1346, 2021.
[34] Google, "Renderscript," https://developer.android.com/training/articles/perf-jni, accessed: 24-03-2025.
[35] Jitpack, "Jitpack.io," https://jitpack.io/, accessed: 24-03-2025.
[36] "Sourcegraph," https://sourcegraph.com/, accessed: 24-03-2025.
[37] "Checksec script," https://github.com/slimm609/checksec, accessed: 24-03-2025.

APPENDIX

A. Bandwidth Optimization

Downloading such a large dataset would require significant bandwidth and excessive storage. However, we manage to reduce this amount by around 15x by downloading only files inside the `/lib/arm64-v8a/` folder instead of the whole APK file. This is made possible by two factors:

- APK files are also valid ZIP compressed archives
- AndroZoo offers an HTTP-based API with support for the `Range` HTTP header, which allows us to download only a portion of the file

ZIP files store metadata about the files they contain in a central directory at the end of the file . This directory contains the offset of each file in the archive. By making a single request to download the central directory, we can then make a request for each individual native library, downloading only the contents of the archive inside the `/lib/arm64-v8a/` folder.

B. Scraping the Sources

Here is an exhaustive list of all the sources from which we obtain ground truth. For each of them, we explain how we download the native libraries and their associated metadata.

1) Maven Metadata: Maven Central is the main repository for Java libraries. It also distributes a large number of Android libraries, both Java and native ones. Native libraries are packaged in files with the `.aar` extension (similarly to jar files, they are normal ZIP archives). ARM64 native libraries are stored inside the folder `jni/arm64-v8a` inside the `.aar` archive. Every Maven package also features a `.pom` XML file that describes the package information.

Downloading all Android native libraries from Maven is a challenging task due to the large number of packages hosted. We divide the process into three steps. First, since Maven Central supports HTTP directory listing, we do a first search in which we iterate over all directories saving only the ones containing an `.aar` file. In the second step, we download the native libraries from each of those archives. However, since `.aar` files also ship various different Android resources (e.g., images, audio samples, and so on), they can grow to a very large size. To limit our downloads to what was strictly necessary, we use the same approach described in Appendix A to only retrieve the contents of the `jni/arm64-v8a` folder from the `.aar` file. Finally, for each archive that contains native libraries, we parse the corresponding `.pom` file and parse the XML tree to extract its metadata. Unfortunately, the `.pom` file does not report the date the package was uploaded, but this is easily fixed by inspecting the modified date of the `.aar` file from the Nginx HTTP directory listing.

2) NPM Metadata: React Native is by far the most popular Android native library bundle (see Table I). In 2016, Facebook stopped using Maven to ship the native libraries for React Native and switched to using `npm` (Node Package Manager). In 2022, however, due to the npm package growing too large in size , Facebook reverted to Maven under a different artifact name. The first version of React Native published on npm instead of Maven was `0.20.0`. The last version published on npm was `0.71.0`.

To retrieve the native libraries that are missing from Maven, we scrape all the stable versions of React Native that are present on npm. To download them, we iteratively query the tarball at URL https://registry.npmjs.org/react-native/-/react-native-<version>.tgz. The tarball contains an `aar` archive from which we extract the libraries in the folder `jni/arm64-v8a`.

3) Flutter Metadata: Flutter is a framework developed by Google for building natively compiled applications for mobile, web, and desktop from a single codebase. An app built with Flutter contains at least two native libraries: the Flutter engine (libflutter.so) and the Dart compiled code (libapp.so). Flutter distributes their native libraries from their archive page [2]. They include the libraries in a zip file, which we download and extract to get the native libraries, marking the date, version and other relevant metadata for each library. We do not collect the Dart libraries, as they are custom for each app.

4) Android NDK Metadata: Apps frequently need to use common shared libraries like `libm.so` (math library) or `libc++_shared.so` (C++ standard library). We found that the vast majority of those are taken from the included set in the Android NDK (Native Development Kit) bundle. To categorize those libraries, we download all versions of the NDK from https://dl.google.com/android/repository/android-ndk-{version}-linux.zip. The Android NDK team offers a page [3] where they list older versions of the NDK available for download. However, we found that the page is not exhaustive and the

[2] https://docs.flutter.dev/release/archive/
[3] https://github.com/android/ndk/wiki/Unsupported-Downloads

majority of the versions are missing. To address this, we try to download all versions of the NDK from version r8 to r30, appending all letters from a to f to include minor releases. Since the date is not included in the download, we use the date of the last modified file in the archive as the release date of the NDK when collecting metadata.

5) Libglog and Libfolly: Google's logging library (`libglog.so`) and Facebook's C++ library (`libfolly.so`) are two of the most popular libraries used in Android apps. React Native relies on them, but statically includes them in their Maven package, without declaring them as dependencies. To address this, we scrape the React Native git repo to extract the version of libglog and libfolly they include by parsing the `gradle.properties` or the `libs.versions.toml` file (depending on the version of React Native). We repeat this process for all versions of React Native from 0.20.0 to 0.71.0.

6) Renderscript Metadata: Renderscript was a high-performance computation engine that was included in the Android SDK. While Google deprecated Renderscript [34] in 2021, it is still frequently used by apps. The renderscript native libraries used by Android apps are prebuilt by Google and can be found in the official renderscript git repository [4]. The relevant libraries are `librsjni.so` and `libRSSupport.so`, and we extract all versions Google ever built by iterating over every commit that modified those files using the command `git rev-list - <path>`.

7) Jitpack Metadata: Jitpack.io [35] is a popular alternative to Maven Central. It allows developers to publish libraries directly from their GitHub repository, and it is the provider chosen by popular libraries like `libucrop.so`. Unfortunately, it is rather challenging to list all packages hosted by Jitpack. First, HTTP directory listing is turned off, which means we can't use the same approach that worked against Maven Central. Secondly, Jitpack had a search feature which could be used to enumerate all hosted packages using prefix-based enumeration techniques. However, at the time of this writing, the search feature is currently non-functional.

To address this, we exploit Jitpack's focus on GitHub repository. By doing a search across all GitHub repositories for URLs starting with `https://jitpack.io/#` using Sourcegraph [36], we managed to gather over 1000 projects that used Jitpack as their distribution platform. Jitpack follows the same distribution mechanism as Maven Central and ships `.aar` archives containing the libraries and `.pom` XML metadata files. Once we locate the `.aar` files, we proceed in a similar fashion to Maven Central. The only difference is in retrieving the date of the upload of the package. Since the `.pom` file does not report any dates, we need to find an alternative solution. One of the main features of Jitpack.io is that, unlike Maven Central, packages are built by Jitpack itself, not by the package developer. As part of this process, Jitpack includes a `build.log` file generated by Gradle to help package developers debug any potential issues. We exploit this feature and download the `build.log` associated with every `.aar` file. In fact, Gradle includes a timestamp in the log, which we can parse to retrieve a close estimate of the publishing time of the package.

[4]https://android.googlesource.com/platform/prebuilts/sdk

Mitigation	On	Off
Fortify	2.32 yrs	3.59 yrs
NX	2.94 yrs	4.15 yrs
Full RELRO	2.93 yrs	3.63 yrs
Stack Canary	2.90 yrs	3.33 yrs

TABLE V: Average age (in years) of libraries compiled with and without the mitigations.

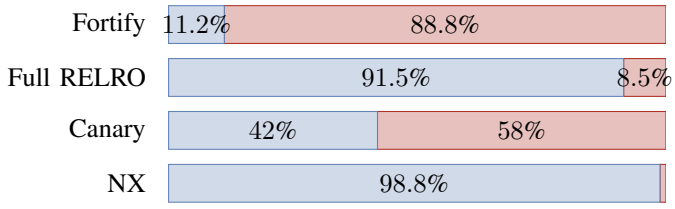


Fig. 5: Prevalence of mitigations in Android apps using native libraries. The color blue represents the percentage of apps using libraries with the given mitigation turned on.

C. Binary Mitigations in Native Libraries

Another useful metric to evaluate the security guarantees of native libraries is to check the binary mitigations that are enabled. We use the `checksec` tool [37] to check the mitigations for each native library.

Figure 5 shows the results. While NX (non-executable stack) is enabled in almost all libraries, the other mitigations are not as prevalent. For Full RELRO, around 13% of libraries do not have it enabled, possibly because it is not turned on by default on modern compilers (Partial RELRO is the default). Surprisingly, around one fifth of the apps rely on libraries without a stack canary, despite the fact that compilers have enabled stack canaries by default for a considerable time. This raises concerns about the prevalence of basic security measures in widely used libraries. The last mitigation analyzed, "Fortify", is not common at all, indicating there is still plenty of room for improvement in the adoption of binary mitigations.

However, we also observe a positive trend, characterized by newer libraries increasingly adopting binary mitigations. Table V shows the average age of libraries based on the presence of binary mitigations, and we notice a clear correlation between the age of the libraries and the presence of mitigations. For example, libraries with "Fortify" enabled are on average more than a year younger than those that do not.