Scratching the Iceberg: Unveiling the Outdated Third-Party Native Libraries in Android Apps

Shiyang Zhang*, Chengwei Liu‡§, Sen Chen†§, Lyuye Zhang‡, Yang Liu‡
*College of Intelligence and Computing, Tianjin University, China
†College of Cryptology and Cyber Science, Nankai University, China
‡College of Computing and Data Science, Nanyang Technological University, Singapore

***Abstract*—Android apps increasingly rely on third-party native C/C++ libraries to deal with low-latency performance and mature functionality, making them a central driver of capability across production deployments. However, opaque binary distribution outside mature package managers impedes discovery, auditing, version tracking, and lifecycle management, allowing technical lag in outdated dependencies to persist and degrade security, compatibility, and reliability. Existing works have investigated the technical lags of third-party libraries in different package manager ecosystems, while Android native libraries are rarely studied due to the lack a comprehensive native library indexing to boost software composition analysis (SCA) tools.**

To this end, by following a greedy and aggressive strategy to identify possible repository sources and collect Android native libraries, we constructed the first comprehensive native library dataset $AndroidNL$ for Android, with over 60K libraries and 292K versions well retained. Our experiments proved its completeness that 85.1% of binaries in real-world APPs can be successfully traced in $AndroidNL$, with 10.1% of the rest suspicious to be not third-party native libraries. Moreover, $AndroidNL$ is also validated to be useful regarding improving native library detection for Android, the experiments show that the state-of-the-art (SOTA) software composition analysis (SCA) tools (i.e., LibRARIAN) can be improved by at least 78.4% on accuracy. Our follow-up studies also highlighted the prevalence and actionable strategy for technical lags on Android native libraries, which could further shed the light on better solutions for the community.

***Index Terms*—Android Native Library, Software Composition Analysis, Tachnical Lag**

I. Introduction

Native C/C++ libraries are integral to modern Android applications because they enable low-latency performance and access to mature functionality to avoid reimplementing the wheels. They are appsprevalently adopted across production apps, shaping both capability and potential fragility [1]. Samhi et al [1] analyzed over 2.8 million Android apps in ANDROZOO and found that 62.9% of them incorporated native code.

The wideadoption of third-party native libraries would introduce significant risk because substantial portions of app functionality depend on external components, increasing the likelihood of vulnerabilities, incompatibilities, and runtime failures. Their persistence within apps fosters technical lag and prolongs exposure to known risks; for example, developers take on average over 500 days to apply native-library security patches, while maintainers release patches in about 55 days [2]. Therefore, accurately identifying, monitoring, and mitigating such delays is essential to improve security and support the long-term stability of the Android ecosystem.

Before undertaking a comprehensive study of technical lag arising from outdated native libraries in Android apps, it is essential to have accurate and reliable detection tooling for Android native libraries. Although many existing studies [3], [4], [5], [6] examine third-party libraries (TPLs) in the Android ecosystem using static analysis, code-clone detection, or machine-learning methods, they primarily target TPLs published and indexed by mature package managers rather than native libraries integrated as binaries within APKs. To the best of our knowledge, Almanee et al. [2] proposed LibRARIAN, the first and, so far, only academic tool to specifically identify Android native libraries via binary similarity analysis. Because it matches binaries against a pre-constructed reference set, the accuracy of LibRARIAN depends critically on the coverage and completeness of that reference corpus. However, LibRARIAN was only equipped with a limited feature dataset constructed from 200 Apps, which limits its applicability to real-world detection at scale. To this end, before studying delays in native-library updates, we construct the first comprehensive dataset of Android native libraries to enable robust detection and to support downstream tasks such as software composition analysis (SCA).

To deal with it, we face the following challenges: **(1) Fragmentation and Lack of a Central Index:** Native libraries are scattered across multiple sources, with no standardized repository or indexing services (like Golang Index [7]). Many are embedded directly in APKs without clear metadata, and dependency resolution varies across build tools, complicating structured aggregation. **(2) Coverage and Dataset Completeness:** The diversity of Android leads to a wide range of native libraries with multiple versions, architectures, and customization levels. Capturing all widely used cases requires extensive crawling, filtering, and deduplication to ensure that the dataset is representative while avoiding redundancy.

Therefore, in this paper, we first proposed an extensive and greedy approach to identify as many potential repositories of Android native libraries as possible and constructed the first comprehensive dataset for Android native library detection. Specifically, for Challenge 1, by systematically analyzing existing ground truths of Android native library importing, we summarized three major types of Android native li-

§ Corresponding authors.

braries based on their importing ways: Directly-Included, Self-Compiled, and Remotely-Integrated Libraries, and collected as many potential repository sources as possible. For Challenge 2, beyond collecting repository sources with explainable evidence, we adopted an extensive searching based on large language models (LLMs) to avoid missing necessary sources. As the result, after collecting all native libraries from identified sources, we constructed a comprehensive native library dataset for Android Apps called $AndroidNL$, with 60,287 libraries and 292,797 versions, to facilitate downstream tasks. Our experiments showed that, against a well-constructed real-world testset of mainstream apps, $AndroidNL$ has significantly covered at least 85.1% of all native libraries in these Apps, with 10.1% suspicious to be Self-Compiled instead of reused from third-party. Moreover, by incorporating LibRARIAN with the feature dataset derived from $AndroidNL$, the identification rate of LibRARIAN has been significantly improved by 78.4%, which also outperformed the SOTA commercial tool BinaryAI [8] by 82.9% on identifiable Native libraries.

Based on this, we further conducted a large scale study to investigate the prevalence of technical lags in real-world Apps based on LibRARIAN with $AndroidNL$, and investigated to what extent these technical lags are supposed to and can be handled by developers. Our empirical analysis revealed that: 1) Almost half analyzed APKs (48.3%) exhibited technical lags, with a mean lag time of 499 days per native library; 2) Explicit evidences showed that these technical lags are more likely to happen on libraries with longer histories and libraries maintained on Maven Central (compared with Google Maven); 3) Mitigating the technical lags that are on minor and patch versions in real-world by directly updating versions actually seldom introduce incompatibility, and doing such updates could result in considerable benefits in terms of outdated lags.

The main contribution of this paper are as follows:

- The biggest contribution of this paper is the construction of $AndroidNL$, by incorporating an extensive and greedy approach, over 60K native libraries and 292K versions have been well collected. $AndroidNL$ can be a comprehensive indexing dataset for Android native libraries, and it could benefit many downstream tasks, such as SCA detection, etc.
- We have also constructed a new feature dataset from $AndroidNL$ called $AndroidNL\_F$ for LibRARIAN, which is validated to largely outperform LibRARIAN with its public dataset and BinaryAI on Android native library identification, by 78.4% and 82.9%, respectively.
- We further conducted an empirical study on the prevalence of technical lags and investigated the possibility of updates for Android native libraries, which revealed some interesting implications to guide further countermeasures.
- We open sourced $AndroidNL$ for public profit [9].

II. Preliminaries

A. Android Native Libraries

Android native libraries play a critical role in the optimization of the functionality and performance of applications, yet their adoption remains largely unregulated, leading to a fragmented and heterogeneous ecosystem. Unlike Java-based Android development, which follows well-documented APIs and standardized frameworks, native libraries can be integrated in a variety of ways, often without stringent guidelines or oversight [10]. This lack of uniformity makes it increasingly difficult to track their usage and potential security risks [2].

B. Related Works

Third-party libraries are a cornerstone of Android application development, prompting extensive research on the identification of third-party library versions. In early time, machine-learning based approaches are widely adopted. For instance, Narayanan et al. [11] developed AdDetect, which uniquely applied semantic detection using machine learning to automatically identify in-app advertising libraries. In contrast, Liu et al. [12] introduced PEDAL, which distinguishes itself by detecting ad libraries even through obfuscated code with its innovative machine classifier. After that, clustering techniques are gradually introduced to detect TPLs in Android Apps. Wang et al. [13] created WuKong, which employs clustering-based preprocessing to filter third-party libraries. Ma et al. [14] implemented LibRadar, focusing on detecting libraries through obfuscation-resistant stable API features. As the progress of program analysis, characteristics of code syntax and semantics are also gradually introduced to enhance the performance. Kalysch et al. [3] proposed an enhanced centroid similarity metric specifically for the detection of native code libraries. Zhan et al. [4] launched ATVHunter, which stands out by using control flow graphs and opcode features to pinpoint vulnerable third-party library versions accurately. Despite these varied and innovative approaches, the primary focus has remained on JVM-based libraries, with rare attention to native libraries.

To the best of our knowledge, there are two papers most related to our scope. Liu et al. [15] is the only existing work that systematically discussed the major building process and tools of Android Apps, while they focused more on the prevalence and practices of configurations of mainstream building tools instead of the sources of native libraries in Android builds. Another most relevant work is conducted by Almanee et al. [2], they introduced LibRARIAN, which employs a novel similarity metric, bin^2sim, leveraging features extracted from library metadata and identifying strings in read-only sections, to identify native libraries and their versions within Android applications. However, only 200 Android Apps, containing in 7.2K binaries, were collected to construct their feature dataset for validation. Moreover, considering that there is no existing indexed repository for Android Native Libraries, it is still far from real-world applicability for LibRARIAN and follow-up software supply chain management. *To this end, we aim to construct the first large scale and comprehensive indexing dataset for Android Native Libraries in this paper.*

III. Approach

In this section, we introduce an extensive and greedy approach to construct the comprehensive dataset for Android Native Libraries. Starting from the well-known Android dataset

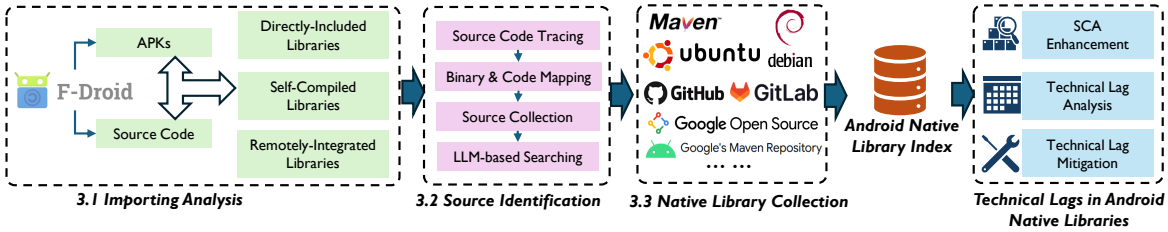

Fig. 1: Overview of this paper

F-Droid [16] that maintains the mapping of Android apps to their source code, 1) we first conduct an empirical study to investigate possible ways of Android native Library importing. Based on that, 2) we identify all possible sources from which the Android native libraries are integrated, and after that, 3) we design a set of pipelines to collect all possible Android native libraries accordingly, to construct the large scale Android native library dataset, as presented in Figure 1.

A. Study on Android Native Library Importing

We first investigate the possible ways in which Android native libraries can be imported into user projects, by inspecting the mapping of source codes and APKs that are well maintained in the F-Droid [16] database.

*1) **Data Preparation**:* F-Droid is an open source Android app repository, it provides Android APKs along with their hosted source code, which strongly supports the subsequent Android native library source analysis. To this end, we first collect APKs that have integrated at least one native libraries. After inspecting the 10,404 APK files that are available on F-Droid, 2,979 APKs, accounting for 28.6% distinct Android Apps are selected out. After that, we further filter out APKs that are with valid source code links, resulting in 2,571 APKs, containing 17,165 Android native libraries, as the final dataset.

*2) **Mapping Analysis**:* To exhaustively identify all possible ways to import Android native libraries, we randomly select 100 APKs for each of the first three authors to label the corresponding importing approaches. The authors are all experienced researchers on software supply chain with at least three years of experience. Specifically, they label the importing approaches by three key factors: 1) whether the Android native libraries are already integrated in source code repositories or not; 2) by which tool the Android native libraries are integrated; 3) what is the evidence of importing corresponding Android native libraries. The detailed statistics of our labeling is available on our website [9].

*3) **Results of Importing Approaches**:* After cross-validation among three authors, we identified three major types of imported Android native libraries.

① **Directly-Included Libraries.** The binaries of Android native libraries are directly included in the source code repositories under certain folders, such as *libs/* or *jniLibs/*.

② **Self-Compiled Libraries.** The source code or links to source code are placed in the source code repositories and compiled to corresponding binaries using Android Native Development Kit [17] or CMake [18] when building Apps.

③ **Remotely-Integrated Libraries** The Android native libraries are downloaded by building tools, namely Gradle [19] and commands like *apt-get install* or *curl*, from package repositories and packed into APK when building Apps.

B. Source Identification for Android Native Libraries

Next, based on the importing approaches we identified above, we design an automated tool $SourceFinder$ to parse each APK and its corresponding source code project and identify possible sources for all 17,165 Android native libraries.

1) $SourceFinder$: Specifically, for each Android APK, $SourceFinder$ first decompresses the APK file and locate all native libraries (i.e., *.so* files). Then, $SourceFinder$ traverses its source code and search evidences for library importing.

① **For Directly-Included Libraries:** During the traversal, $SourceFinder$ first identifies all native libraries that are explicitly retained in the source code project by checking the file extension (i.e., *.so*). Specifically, considering that some directories, such as */lib* and */jniLibs*, may contain multiple binary files of the same library for different system architecture, $SourceFinder$ records these names and locations only once for further mapping with native libraries identified in APKs.

② **For Self-Compiled Libraries:** $SourceFinder$ first identifies all shell scripts that have executed git download command, as exampled in Listing 1. Then, $SourceFinder$ downloads such external files as submodule for further analysis along with the original source codes. After supplementing all the potentially involved source codes, $SourceFinder$ locates files that configure the bindings between Android native library source code and generated binaries. Specifically, we focus on *Android.mk* (for Android NDK) and *CMakeLists.txt* (for CMake) and identify the corresponding configurations. The *CMakeLists.txt* usually declares the compilation of native libraries by keywords *add_library* and *SHARED*, as exampled in Listing 2. The *Android.mk* configures compiling native libraries by keywords *LOCAL_MODULE* and *include $(BUILD_SHARED_LIBRARY)*, as exampled in Listing 3. To this end, $SourceFinder$ collects the configured names and locations of corresponding source code throughtout the source code projects as records for further mapping to native libraries.

③ **For Remotely-Integrated Libraries:** $SourceFinder$ identifies two folds of possible configurations in source code repositories. For Android native libraries that are introduced by Gradle, $SourceFinder$ inspects the specific configuration file *build.gradle* to identify the imported packages. Specifically, $SourceFinder$ first inspects the *build.gradle* in the root

```
git clone https://gitlab.linphone.org/BC/public/linphone-sdk.git
cd linphone-sdk
git checkout "${BRANCH}"
git submodule sync --recursive
```

Listing 1: Example of downloading remote source code for self-compiled libraries in *org.simlar* [20].

```
add_library(
        fabricjni
        SHARED
        ${fabricjni_SRCS}
)
```

Listing 2: Example of declaring Android native libraries in *CMakelist.txt* in *com.github.meypod.al-azan* [21]

```
LOCAL_MODULE := main

LOCAL_SRC_FILES :=
  $(SDL_PATH)/src/main/android/SDL_android_main.c \
        agg-2.5/src/agg_arc.cpp

LOCAL_SHARED_LIBRARIES := SDL2 SDL2_image SDL2_mixer
include $(BUILD_SHARED_LIBRARY)
```

Listing 3: Example of declaring Android native library in *Android.mk* in *se.traffar.dot_race* [22]

```
//In the build.gradle file in the root folder.
allprojects{
    repositories {
        google()
        mavenCentral()
        maven { url 'https://jitpack.io' }
        jcenter()
    }
}
// In the build.gradle files in submodule folders
dependencies {
    implementation
      'com.facebook.fresco:animated-gif:2.5.0'
    implementation(libs.jxl.coder)
}
```

Listing 4: Example of importing Android native libraries in *build.gradle* in *com.podverse.fdroid* [24], [25]

```
[versions]
...
jxlCoder = "2.4.0.7"
...
[libraries]
jxl-coder = { module = "io.github.awxkee:jxl-coder",
version.ref = "jxlCoder" }
...
```

Listing 5: Example of TPL declaration in *libs.version.toml* in *ru.tech.imageresizershrinker* [26]

folder, then identifies and records all possible sources configured in *allprojects* for further collection of Android native libraries (e.g., line 2∼9 in Listing 4). Next, *SourceFinder* visits the *dependencies* sections in all *build.gradle* in submodules, to identify and record the GAV (i.e., the group id, artifact id and version number) of all configured dependency packages (e.g., line 14 in Listing 4). After that, considering that after Gradle 7.0, Version Catelogs [23] are introduced, *SourceFinder* also identifies the *libs.version.toml* file, as exampled in Listing 5, to trace the package names declared in *build.gradle* back to their original GAVs (i.e., line 14 in Listing 4). After that, the identified package names and repository sources are all logged by *SourceFinder* for further mapping analysis with Android native libraries. For Android native libraries that are integrated from platform-related libraries downloaded during the build phase, *SourceFinder* specifically inspects all shell scripts to find commands that download packages, including *apt-get*, *wget*, and *curl*. After that, *SourceFinder* record the identified package names and corresponding registries for further mapping analysis.

④ **Source code & Binary Mapping.** After collecting all these clues in source code project that can indicate the integration of Android native libraries, *SourceFinder* then maps these clues back to the native libraries (i.e., *.so* files) in APKs to ensure that the source of all native libraries are properly identified. 1) For Directly-Included Libraries, *SourceFinder* directly matches them back to native libaries in APK by names and hashes. 2) For Self-Compiled Libraries, *SourceFinder* maps the source code locations to native libraries by the defined names in configuration files (e.g., *fabricjni* in Listing 2 and *main* in Listing 3). 3) For Remotely-Integrated Libraries, *SourceFinder* downloads the corresponding packages from remote repositories, and decompresses them to retrieve the binary files, then matching them with those native libraries identified in APKs by LibRARIAN. After this, we can obtain the detailed mappings between native libraries in APKs and the corresponding source code locations or remote addresses.

⑤ **LLM-based Extensive Searching.** After mapping all identified Android native libraries in APKs, there could still be native libraries that are not mapped to any existing import clues. To ensure that we can retain as many possible sources as we can, *SourceFinder* is designed to conduct an extensive search to find possible sources for those missed cases, based on LLM. Note that many binaries in APKs may not be native libraries (e.g., private tools), therefore, the LLM-based searching only aims to avoid neglecting well-known third-party binaries instead of ensuring all identified binaries being able to be mapped to known libraries.

Specifically, for each missed native library, *SourceFinder* asked the GPT-4o to provide possible source links, with additional information, such as APK name and their category in APKCombo. For each case, *SourceFinder* queried the GPT-4o for each prompt in Figure 2 for three times with temperature set to 0 to ensure the stability of the results. We merged all results of the three prompts and filtered out those links of registries or repositories as the final result. After this, we combined all the collected data sources with previously identified ones as the candidate sources to collect Android native libraries.

*2) **Results**:* We applied our tool to all the 2,571 APKs and 17,165 native libraries obtained in Section III-A1. The exper-

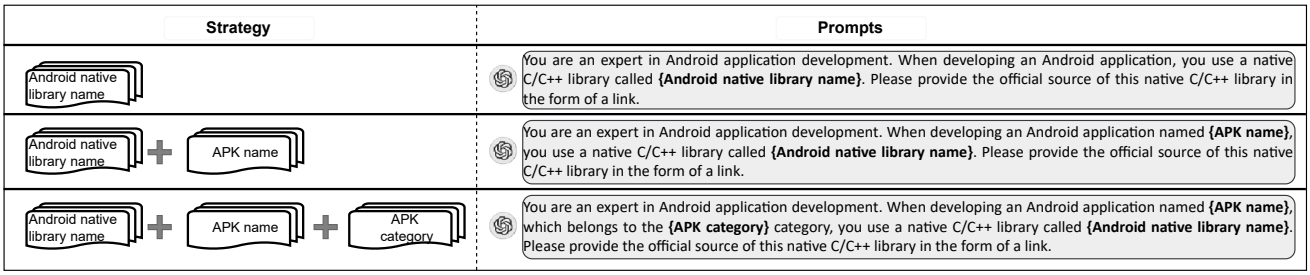

Fig. 2: GPT Prompts

imental results show that, in total, 15,666 out of these 17,165 native libraries in APKs (91.3%) are successfully traced to their sources with confirmed evidences (i.e., successfully mapped in Step d). In detail, as presented in Table I, 96 (0.6%), 6,344 (37.0%), and 9,226 (53.7%) of them are directly-included, self-compiled, and remotely-integrated libraries, respectively. As for the rest 1,499 unrecognizable native libraries (8.7%), the extensive searching (i.e., Step ⑤) resulted in 363 web links of 244 libraries. These links covered 62 unique web domains, within which only 5 web domains appeared more than 5 times in all results. Subsequently, we conducted an manual analysis of the cases that still failed and found that such missing are due to customized library-importing methods of specific frameworks. For instance, through manual analysis, we found that, in projects that are with Qt architecture [27], QtCreator will automatically include some Qt dependencies during compilation, and only some developers specified them in *README* as prerequisites. Considering that exploring all such mechanisms may require substantial manual efforts, and the proportion of missing cases currently is relatively small, we neglected these cases for now in this work.

Based on these results, we further summarized the collected sources to determine the native library collection strategies.

For directly-included libraries, since there are no hints about the sources of them, we are unable to trace where they are from. Instead, we rely on LibRARIAN to match them with other collected native libraries to find their identities.

For self-compiled libraries, although some of these libraries are from external source code repositories and integrated after local compilation, it does not mean we can directly collect the corresponding binary files from these repositories. Moreover, considering that some Android native libraries are directly kept in source code repositories, we add these source code repositories as possible sources for further collection, such as GitHub, GitLab, and Gitdab.

For remotely-integrated libraries, we further investigated the distribution of their corresponding sources, as presented in Figure 3. In total, 9 repositories for TPLs are included. Specifically, Google Maven[28] and Maven Central[29] are the top-2 sources of Android native libraries from TPLs. Jcenter and Bintray also appears in a large proportions of gradle files, but they have been shutdown and transformed to Maven Central [30], [31]. The repositories of operating systems are also major sources of a large portion of remotely-integrated

TABLE I: **Source Code Analysis Results**

Scenarios	Type	Libraries
Directly-Included	/	96
Self-Compiled	Local Source Code	4,795
	External Source Code	1,549
Remotely-Integrated	Third-Party Libraries	8,675
	Operating System Components	551
Mismatched	/	1,499

libraries, such as Ubuntu[32] and Debian[33]. Jitpack[34] is not an independent central repository that keeps third-party libraries, configuring it as sources in Android development is a common practice to simplify the procedure to integrate GitHub repositories. Sonatype[35] is an alternative sources for most Maven libraries. The Bintray [36] and CommonsWare [37] are individual sources of specific software with small groups of TPLs, and Clojars [38] is a repository for Clojars packages, considering that Clojars packages are usually for web apps, and seldom contain jars for Android.

As for the supplement sources from extensive searching, we identified 62 unique web domains from the searching results. After excluding those are not actually repositories for Android native libraries (e.g., issuetrackers [39], online analyzers [40]), we obtained 4 major web domains (GitHub, *gitlab.ujaen.es*, Google Source, and *pkgs.org*) that could be sources of Android native libraries. Considering that TPLs on *pkgs.org* are mainly OS packages, we only additionally include the Google Source as a new potential source.

To this end, we collected a set of sources that maintain the mainstream Android native libraries, including Git-repositories (i.e., GitHub, GitLab, Gitdab, Google Source, etc.), OS repositories (i.e., Ubuntu and Debian, etc.), Maven-alike repositories (i.e., Maven Central and Google Maven, etc.).

C. Native Library Collection

After identifying these possible sources, we collected these Android native libraries correspondingly.

- **OS Package Repositories:** We mainly collected Android native libraries from Ubuntu and Debian repositories [32], [33], which share the same structure, organizing packages into subdirectories based on their names. Each package directory contains multiple versions of binary package files (*.deb*), source packages (*.tar.xz*), and Debian source control files (*.dsc*). Since Android native libraries are embedded within *.deb*

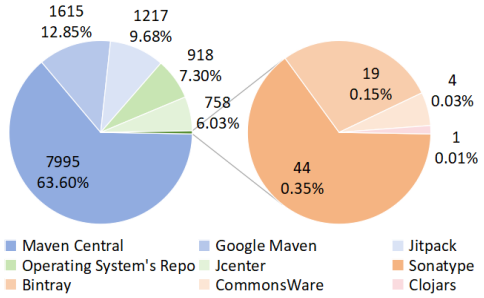

Fig. 3: Date Source Distribution

TABLE II: **Android native Libraries Dataset Distribution**

Data Source	#Libraries	#Version
Debian	32,261	120,993
Ubuntu	8,040	48,373
Maven Central	18,358	112,849
Google Maven	69	4,872
Google Source	617	2,480
Git Repositories	942	3,230

archives alongside other files, we first recursively downloaded all *.deb* files, then decompressed and extracted the native libraries by BinWalk [41]. We further filtered relevant files and recorded metadata, including package name, version, compiler architecture, and release time. This approach resulted in the collection of 48,373 Android native libraries from Ubuntu and 120,993 from Debian.

- **TPL Repositories:** This type of data source mainly includes Maven Central [29] and Google Maven [28], which store a variety of software artifacts. However, identifying Android native libraries is challenging as they are embedded within *.aar* files alongside other resources. To address this, we recursively downloaded, extracted, and filtered native libraries from *.aar* packages and recorded relevant metadata such as package name, version, and release timestamp. This process resulted in the collection of 112,849 Android native libraries from Maven Central and 4,872 from Google Maven.
- **Source Code Repositories:** Google Source [42], GitHub [43], GitLab, and other Git-based repositories are mainly for the version control of source code projects, therefore, it is not straightforward to directly collect all possible Android native libraries. Instead, we conducted customized searching on each platform, specifically, we searched with keywords on file types (i.e., **.so* and **.apk*) to filter those that are suspicious to contain Android native libraries. Moreover, due to their diverse repository structures and access limitations—including multiple branches and tags, as well as API restrictions on retrieval—these platforms also pose challenges for data collection. To overcome these issues, we employed a combination of automated downloads, web crawlers, extraction and filtering, and finally collected 2,480 Android native libraries from Google Source and 3,230 from Git Repositories.

Overall, we collected a total of 60,287 native libraries with 292,797 versions from various sources and built a complete dataset $AndroidNL$ of native libraries along with their data sources and relevant metadata. The distribution for libraries and versions in each data source are presented in Table II.

IV. Experiments

In this section, we describe the experiments we conducted and focus on the following two research questions (RQs):

- **RQ1: Completeness of** $AndroidNL$**.** How complete is $AndroidNL$ in terms of Android native libraries?
- **RQ2: Improvement for SCA.** To what extent can $AndroidNL$ enhance existing SOTA SCA tools in terms of detecting third-party Android native libraries?

To evaluate $AndroidNL$, we conducted experiments in the following two scenarios: ① Evaluating the coverage of $AndroidNL$ in a large-scale *real-world dataset*. ② Assessing the extent to which $AndroidNL$ enhances the performance of existing SCA tools for native libraries in Android Apps. To evaluate the effectiveness of SCA tools with our data, we constructed the feature dataset based on $AndroidNL$ for LibRARIAN, which is the SOTA SCA tools for Android native libraries based on code clone detection techniques. For better presentation, we denote the feature dataset as $AndroidNL\_F$.

A. RQ1: Completeness of $AndroidNL$

Given the cutting edge performance of LibRARIAN, we use it to verify the completeness of $AndroidNL$. We planned to extract Android native libraries from a large-scale real-world Android apps and trace the source of these native libraries in $AndroidNL$ using LibRARIAN.

DataSet. APKCombo [44] hosts a vast collection of APKs that largely mirror the most popular and widely used applications on mainstream App Stores, such as Google Play, making it a representative repository of mainstream Android apps. APKCombo offers 24 categories of apps and a "top-popular" list for each category. Considering the completeness and timeliness of the projects, we downloaded all APKs released over the past five years for Android projects listed in the "top-popular" list across all these 24 categories. In total, we downloaded 14,734 APKs from 477 Android projects and extracted 150,914 native libraries (i.e., *.so* files) from them. To avoid the rarely used instance and focus on mainstream libraries, we retained only those found in more than 10 Android projects, specifically totaling 63,008 native libraries. Among these 63,008 native libraries, features were successfully extracted from 61,325 of them by LibRARIAN, denoted as *Real-world dataset*.

Result: We compared *Real-world dataset* against $AndroidNL\_F$ by LibRARIAN. The experimental results showed that 52,194 binaries, out of 61,325 (85.1%) can be successfully mapped to $AndroidNL\_F$, which means these binaries are well covered as Android native libraries in our $AndroidNL$ dataset. We further investigated the sources of successfully matched cases in Table III. Statistics show that Maven Central [29] plays the most significant role in $AndroidNL$, followed by Google Maven [28] and GitLab [43], illustrating the preference on these sources. Subsequently, we conducted an in-depth analysis of all instances that fail to achieve successful matching:

TABLE III: **Initial Sources of Successfully Matched Android native Libraries**

Source	MC*	GM*	Gitlab	Github	Gitdab
Number	37,021	6,740	4,744	1,863	1,826
Proportion	70.9%	12.9%	9.1%	3.6%	3.5%

MC* represents Maven Central. GM* represents Google Maven.

• **Self-Compiled Android Native Libraries.** As discussed in Section III-A2, Android native libraries from F-Droid dataset revealed self-compiled 37.0% of libraries. Based on this observation, we hypothesized that a similar proportion of self-compiled C/C++ libraries may exist within APKCombo. However, given that APKCombo does not provide access to the source code of Android projects, identifying such self-compiled libraries required an alternative approach. Specifically, we compared the SHA-256 hashes and filenames of untraceable Android native libraries with those of self-compiled ones in Section III-A2. Out of the 9,131 untraceable Android native libraries, 3,039 exhibited identical filenames and SHA-256 hashes to self-compiled libraries, while 3,349 shared identical filenames but differed in their SHA-256 hashes, indicating in total 6,388 (10.1%) libraries were likely introduced through self-compilation.

• **Incomplete Version Collection Due to Data Source Updates.** In certain instances, the identification results of Android native libraries with the same filename, originating from different versions of the same Android project, were inconsistent. Considering that in Android applications, the use of Android native libraries typically exhibits a certain level of continuity across versions, i.e., in most cases, different versions of an APK tend to reuse identical or similar native library, we hypothesized that these failures may be attributed to incomplete version collection rather than the complete absence of the data source. This data loss may be caused by updates to the data sources. Among the remaining 2,743 failed cases, 446 exhibited the aforementioned phenomenon.

Finding 1: In the *Real-World dataset* we constructed, 52,194 binaries (85.1%) identified in Apps from APKCombo can trace the source within our collected $AndroidNL$ dataset, and the majority of the mismatched cases (10.1%) are likely introduced through self-compilation, verifying the completeness of our Android native library dataset.

B. RQ2: Improvement for SCA

Given the large coverage of $AndroidNL$, we further assess the extent to which $AndroidNL\_F$ enhances the performance of existing Android native library version identification tools with two selected baselines. These baselines include SOTA model and leading commercial tool, ensuring that the experimental results are convincing and comprehensive. The baselines including our own enhanced approach are as follows:

• **Baseline 1: LibRARIAN model with its public dataset.**
• **Baseline 2:BinaryAI [8] platform.**
• **Our approach: LibRARIAN with $AndroidNL\_F$.**

Due to the upload file size and API limitations on the BinaryAI platform, we had to use a smaller dataset for the BinaryAI experiment. Therefore, to compare with Baseline 1, we applied the pre-constructed *Real-world dataset*, and to compare with Baseline 2, we randomly selected 1,000 Android native libraries from the *Real-world dataset* to create a smaller dataset. Our enhanced approach is implemented to verify whether $AndroidNL\_F$ can enhance both the SOTA model and the leading commercial tool.

LibRARIAN: We conducted comparative experiments on *Real-world dataset*, using both the origin public feature dataset of 825 native libraries provided by LibRARIAN and $AndroidNL\_F$. The experimental results demonstrated an recognition increase from 4,121 libraries (6.7%) to 52,194 libraries (85.1%) beyond LibRARIAN's original dataset, and all libraries identified by LibRARIAN with its public dataset can be successfully identified by LibRARIAN with $AndroidNL\_F$, indicating that $AndroidNL\_F$ can significantly enhances the capability of LibRARIAN since LibRARIAN matches binaries against a pre-constructed reference set, the accuracy of LibRARIAN depends critically on the coverage and completeness of that reference corpus. Moreover, while the LibRARIAN public dataset does not specify the source of each data, our dataset provides detailed annotations on origins of native Android libraries, offering a foundation for related tasks such as technical lag analysis in the future.

BinaryAI: For BinaryAI, we evaluated its official API and found that, among the 1,000 Android native libraries, 41.2% yielding a total of 1,351 component links. After manual inspecting, we only identified 151 official links corresponding to the native libraries, and all of them are from GitHub. In comparison, we ran *our enhanced approach* to trace the origins of these 1,000 Android native libraries and successfully identified 980 of them with detailed original sources, achieving a traceability success rate of 98.0%. Moreover, *our enhanced approach* successfully identified 150 out of the 151 succeed cases of BinaryAI, and even for the missed one, *our enhanced approach* actually has identified its source but this result is rejected due to its own threshold of required similarity. These comparative experiments led us to the conclusion that integrating $AndroidNL\_F$ with LibRARIAN significantly improves the recognition rate of Android native libraries from 15.1% to 98.0%. We believe our dataset has a distinct advantage because it is specifically focused on Android native libraries, whereas BinaryAI targets a broader range of domains, resulting in less effective performance.

Finding 2: Regarding Android native library identification, compared to the SOTA and leading commercial tool, the use of $AndroidNL\_F$ results in an improvement of 78.4% and 82.9% in recognition rate respectively, indicating the outstanding usefulness of $AndroidNL\_F$.

V. Empirical Study of Outdated Native Libraries

The above experiments have demonstrated that $AndroidNL$ can significantly improve the detection of third-party Android

native libraries in practice, based on it, we further investigate the prevalence and severity of technical lags of outdated native libraries in Android Apps, and to what extent they can be relieved, by answering the following two research questions:

- **RQ3: Technical Lag Analysis.** How prevalent are technical lags existing in Android APKs?
- **RQ4: Technical Lag Mitigation.** To what extent can Android developers automatically update embedded native libraries in APKs without introducing compatibility issues?

A. RQ3: Technical Lag Analysis

In Android apps, the technical lag refers to issues of not using the latest versions of dependencies, especially the native libraries, potentially leading to challenges in performance, security, and compatibility. We conducted a large-scale empirical study based on RQ1 result to analyze the prevelance of technical lag in mainstream Android Apps.

*1) **Dataset**:* To systematically investigate the technical lags of Android native libraries, we conducted a detailed investigation of the traced data sources in RQ1 results, as presented in Table III. In this RQ, we will analyze the technical lag phenomenon as comprehensive as possible, provided that the data sources can provide accurate library release times. Considering Git repositories are not always considered as publishing platforms, and directly using their time could be unfair and amplify the technical lags, We have to exclude libraries related to Git in this RQ. In the end, we chose the remaining native libraries from Google Maven and Maven Central, luckily they accounted for 83.8% of *Real-world dataset*, as the dataset for this RQ, also known as *Real-world dataset for Technical Lag*.

As for the calculation of technical lags, APKCombo provides release timestamps for each published APK, while Maven Central and Google Maven provide last modification timestamps for each released library component. For a given case, if APK did not include the latest version of an Android native library at its publish time, we considered this case to exhibit technical lag, and in this case, we calculated the time interval between the APK release time and the published time of the included Android native libraries as the value of technical lag. Note that we did not follow traditional time-based technical lag [45] to calculate the time interval between current date and release time of the version of included Android native libraries, because the APKs are usually unmodifiable after they are published, and usually developers update APKs by directly releasing a new version. In this case, we focus on the lags of versions that developers overlooked but are supposed to be aware when publishing APKs.

*2) **Lag Distribution over Native Libraries**:* Our analysis indicated that among all the 43,761 Android native libraries used in 5,699 APKs, 21,147 (50.3%) cases from 2,864 (48.3%) APKs exhibited technical lag, demonstrating that technical lag is a prevalent phenomenon in Android apps.

We further investigated the distribution of technical lag time in Android native libraries. For each APK, we defined its total lag time as the sum of the technical lag times of all native libraries it uses. [45]. Subsequently, we analyzed the technical lag from both the APK and the individual native library perspective.

① **Lag Distribution regarding APKs:** First, the distribution of APKs across different technical lag intervals is shown in Figure 4. The results indicated that technical lag is prevalent for more than one year, suggesting that developers of Android applications have largely neglected the selection and updating of Android native library versions, leading to severe technical lag issues. Next, we analyzed the average technical lag for APKs across app store categories, the results revealed that APKs under *Event* category exhibit the longest average delay, whereas *entertainment* APKs experience relatively smaller technical lag.

② **Lag Distribution regarding Libraries:** The average technical lag for each native library was calculated based on its usage across APKs (Figure 5). The results showed that while most libraries exhibit relatively concentrated lag durations, some experience significantly higher technical lag. The median lag was 511 days, with the first and third quartiles at 295 and 783 days, respectively, indicating that most native libraries lag by over a year. This highlights the need for developers to make informed library choices and ensure timely updates.

*3) **Lag Distribution over Semantic Versions**:* We also investigated the distribution of lagged versions, as categorized into three types based on SemVer [46]: 1) Major version update lag, where the delay crosses major versions and the incompatible changes are allowed. 2) Minor version update lag, where the delay occurs across minor versions only and backward-compatible functionality is added. 3) Patch version update lag, where neither major nor minor version updates are required and only compatible changes are allowed. It implies that minor and patch version lag has the potential for automated upgrades. Among all the 21,147 lagged cases, 34.4% required a major version upgrade and 41.3% for the minor, whereas the remaining 24.3% required the patch upgrade only, as presented in Figure 7, indicating that 65.6% could have been automatically upgraded. In Section V-B, we will explore the feasibility of automated upgrades for them.

*4) **Influencing Factors of Technical Lag**:* After validating the presence of technical lag in Android native libraries, we investigated the factors influencing the lag. Specifically, we analyzed the impact of library usage frequency, library release time, APK release time, and library source on technical lag. Pearson [47] and Spearman [48] correlation metrics were used to quantify these influences (-1: perfect negative and +1: perfect positive).

The experimental results show that the usage frequency of Android native libraries has minimal impact on technical lag, with Pearson and Spearman correlation coefficients of 0.05 and 0.11, respectively, indicating no significant correlation. Similarly, the release time of APKs has little effect, as evidenced by Pearson (-0.11) and Spearman (-0.24) coefficients, neither of which suggests a strong relationship. In contrast, the release time of Android native libraries strongly influences

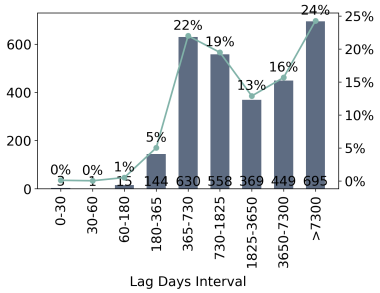

Fig. 4: APK Lag Interval

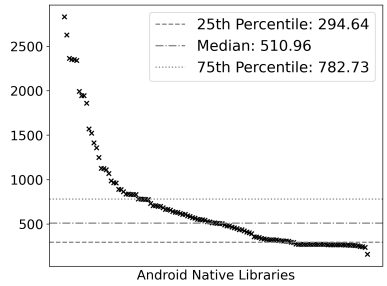

Fig. 5: Library Lag Distribution

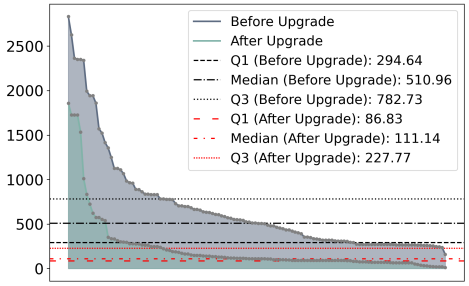

Fig. 6: Effectiveness of version updates.

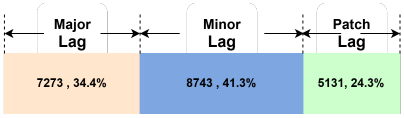

Fig. 7: Lag Distribution over Semantic Versions

TABLE IV: **Lag Distribution from Different Sources**

Source	lag(days) count				median	mean
	0-180	180-365	365-730	730-3607		
all	2,142	4,748	6,997	7,260	533	739
MC*	2,132	4,282	4,520	6,944	530	774
GM*	10	466	2,477	316	543	546

MC* represents Maven Central. GM* represents Google Maven.

technical lag, with Pearson and Spearman coefficients of 0.94 and 0.86, respectively. This suggests that libraries with longer histories are more prone to technical lag issues, this is reasonable as there could be more than one major version tracks maintained simultaneously. However, such delays of adopting new versions could also lead to the gradual delay of adopting new features and better performance.

To evaluate the impact of a library's source on technical lag, we analyzed the lag of native libraries from different sources. The experimental results presented in Section V-A4 indicate that Android native libraries from Maven Central experience a maximum technical lag of 3,607 days, with the mean as 774 days and the median as 530 days. In contrast, libraries from Google Maven have a lower maximum lag of 1,516 days, with a mean lag of 546 days and a median of 543 days. Comparatively, Android native libraries from Google Maven show significantly shorter technical lag than those from Maven Central in terms of maximum and average delay. This disparity may stem from differences in data maintenance quality between sources. Developers are thus encouraged to prioritize well-maintained sources to minimize technical lag.

Finding 3: The technical lag of native libraries is pervasive within the Android ecosystem, with a mean lag time of 739 days. Specifically, these lags are more commonly happened on libraries that are with longer history; libraries released on Maven Central, compared to Google Maven which is with more regulated maintenance, also relatively have longer lags. Android developers are suggested to consider these factors when selecting libraries.

B. RQ4: Technical Lag Mitigation

The prevalence of technical lags is known to amplify the risks of security exposure, incompatibilities, and operational instability in user projects, and it also inflates maintenance costs, erodes compliance and vendor support, and slows delivery of new capabilities, which could ultimately reducing engineering velocity, user trust, and competitiveness. After demonstrating the prevalence of technical lag in RQ3, in this RQ, we aim to investigate to what extent can Android developers avoid the technical lags on native libraries, and explore automated and applicable upgrade strategy to mitigate the risks of technical lags.

Data Preparation. There, we first constructed another dataset with precisely labelled library versions. To achieve this, we selected data from GitHub [43], where project source code provides a reliable reference for exact library versions. To ensure data comprehensiveness while limited by GitHub API return value quantitative, we used the GitHub API to collect 1,000 repositories for each of the 20 star count ranges, totaling 20,000 repositories. Filtering for those with APKs in their releases yielded 240 repositories. After manual analysis, we excluded repositories lacking Android native libraries or exact version information, and take the latest release tags for each repository, ultimately collecting 261 libraries from 68 APKs.

*1) **Compatible Version Updates**:* We first look into the proper version update strategies for Android native libraries that would not introduce incompatibility. For these 68 Android projects, we manually compiled their source code hosted on GitHub [43] and upgraded the native libraries used sequentially. To ensure the compatibility, only minor and patch upgrades have been attempted.

We assessed post-upgrade compatibility using two complementary approaches:

① **The Monkey Test.** Monkey programmatically emits pseudo-random UI events (taps, swipes, keystrokes, rotations, back/home) to exercise diverse paths; a run is deemed a pass if no uncaught exceptions, ANRs, or fatal crashes occur and the app remains responsive until the event budget is exhausted. We set the number of events to 5000 and the seed value to 12345 for reproduction.

② **The SceneDroid Test [49].** SceneDroid combines guided exploration, state fuzzing, and indirect launching (e.g., deep links/intents) to construct a fine-grained Scene Transition Graph (SceneTG) whose nodes/edges represent GUI states

and feasible transitions. We build SceneTGs before and after the upgrade and compare them using overlap of reachable nodes/edges and local neighborhood similarity to detect navigation changes. SceneDroid analysis is restricted to apps whose baseline APK yields a well-formed SceneTG to avoid confounding from exploration incompleteness.

Results. Across 68 Android projects, 31 (45.6%) were already on the latest minor versions. Of the remaining 37, upgrading to the latest minor versions resulted in 31 Monkey passes (83.8%), suggesting limited regression risk under randomized interaction. For SceneDroid, 25 apps also passed post-upgrade with exactly same SceneTGs and no material divergences in navigation structure. As for the failed 12 cases, after consulting the authors of SceneDroid, the failures are due to limitations of the re-packaging tool they used during the process. These findings indicate that an automated upgrade-to-latest-minor version strategy can largely reduce technical lag while rarely introducing compatibility regressions detectable by stress testing or dynamic GUI-equivalence analysis, which could be considered as solutions for developers to upgrade legacy native libraries in their Apps.

*2) **Large-Scale Trial on the real-world dataset**:* Subsequently, we followed the previous same upgrade strategy to perform automated upgrades for the Android native library in RQ2. The experimental results in Figure 6 illustrate the impact of our automation upgrade strategy, with the blue and green areas representing technical lag distributions before and after the upgrade, respectively. The results show a significant reduction in technical lag. Specifically, the 25th percentile decreased from 295 to 87 days (208-day reduction), the median decreased from 511 to 111 days (400-day reduction), and the 75th percentile decreased from 783 to 228 days (555-day reduction). These improvements highlight the effectiveness of our approach in mitigating technical lag, benefiting the overall Android ecosystem.

Finding 4: The automatic Android native library upgrade strategy, namely, upgrading to the latest stable version with the same major version, has been validated. We applied this upgrade strategy to Android native libraries with the phenomenon of technical lag in RQ2 and reduce the 25th percentile lag time by 208 days, the median lag time by 400 days, and the 75th percentile lag time by 555 days.

VI. Limitations and Threats to Validity.

Our study has several limitations, though their influence is mitigated by design choices and empirical validation. **1) Sample Selection:** When constructing $AndroidNL$, we chose to analyze potential data sources from the APKs published on F-Droid. Due to the open-source nature of F-Droid, this may have overlooked the use of native libraries in closed-source Android applications to some extent. However, in the experimental section, we verified the coverage of $AndroidNL$ in the real-world Android ecosystem through data from APKCombo, thus demonstrating the rationality of our approach and the selection of F-Droid. Similarly, we analyzed APKs from popular apps on APKCombo [44] (2019–2024), which may overlook older or less popular apps. However, since popular apps tend to be well-maintained and widely used, they provide a representative view of real-world dependency management. **2) Data Collection Constraints:** GitHub's API restrictions [43] limited our ability to retrieve all Android native libraries. To mitigate this, we included libraries referenced in build.gradle files (Section III-A2), ensuring that our dataset still captures the most commonly used dependencies. **3) Comparison Constraints:** Due to the limitation of the binaryai API, in RQ2, our enhanced approach only used 1000 cases for comparison with Baseline 2 without conducting full comparison, which may lead to doubts about the significance of the results. However, considering that 1000 cases were randomly selected and the final results showed a large gap (an improvement of 82.9% in recognition rate), the conclusion can be considered as valid. **4) Automated Upgrade Scope:** Our upgrade strategy limits recommendations to the same major version to avoid breaking changes. While cross-major upgrades are sometimes feasible, they often require manual intervention due to potential incompatibilities, making them impractical for large-scale automation. The current approach thus provides a realistic and applicable upgrade strategy.

VII. Conclusion

In this paper, we aimed to investigate the outdated native libraries in Android Apps. To achieve this, we constructed the first comprehensive indexing dataset for Android native libraries $AndroidNL$, by following a greedy and aggressive strategy to identify repository sources and collect native libraries as many as we can. $AndroidNL$ is proved to achieve high coverage (at least 85.1%) on binaries used in real-world Android Apps, and can effectively improve SOTA SCA detections with at least 78.4% recognition rate improvement. After that, we investigated the prevalence and possible solutions for technical lags on Android native libraries, which shed light on better countermeasures for the open source community.

VIII. Data Availability

The constructed $AndroidNL$ other experiment datasets can be found at our website [9].

Acknowledgments

This research is partially supported by the National Natural Science Foundation of China (No. 62472309); the National Research Foundation, Singapore, and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG4-GC-2023-008-1B); the National Research Foundation Singapore and the Cyber Security Agency under the National Cybersecurity R&D Programme (NCRP25-P04-TAICeN); and the Prime Minister's Office, Singapore under the Campus for Research Excellence and Technological Enterprise (CREATE) Programme. Any opinions, findings and conclusions, or recommendations expressed in these materials are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore, Cyber Security Agency of Singapore, Singapore.

REFERENCES

[1] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein, "Jucify: A step towards android code unification for enhanced static analysis," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1232–1244.

[2] 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 *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1347–1359.

[3] A. Kalysch, O. Milisterfer, M. Protsenko, and T. Müller, "Tackling androids native library malware with robust, efficient and accurate similarity measures," in *Proceedings of the 13th international conference on availability, reliability and security*, 2018, pp. 1–10.

[4] X. Zhan, L. Fan, S. Chen, F. We, T. Liu, X. Luo, and Y. Liu, "Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1695–1707.

[5] Y. Wu, C. Sun, D. Zeng, G. Tan, S. Ma, and P. Wang, "{LibScan}: Towards more precise {Third-Party} library identification for android applications," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 3385–3402.

[6] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen, "Detecting third-party libraries in android applications with high precision and recall," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 141–152.

[7] "Go modules services," 2023, (Accessed on 03/27/2023). [Online]. Available: https://proxy.golang.org/

[8] "Binaryai," 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://www.binaryai.cn/

[9] "Android native library," [Online; accessed 2025-03-14]. [Online]. Available: https://sites.google.com/view/hi-library

[10] A. Ruggia, A. Possemato, S. Dambra, A. Merlo, S. Aonzo, and D. Balzarotti, "The dark side of native code on android," *ACM Transactions on Privacy and Security*, 2022.

[11] A. Narayanan, L. Chen, and C. K. Chan, "Addetect: Automated detection of android ad libraries using semantic analysis," in *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*. IEEE, 2014, pp. 1–6.

[12] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient privilege de-escalation for ad libraries in mobile apps," in *Proceedings of the 13th annual international conference on mobile systems, applications, and services*, 2015, pp. 89–103.

[13] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Wukong: A scalable and accurate two-phase approach to android app clone detection," in *Proceedings of the 2015 international symposium on software testing and analysis*, 2015, pp. 71–82.

[14] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: Fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th international conference on software engineering companion*, 2016, pp. 653–656.

[15] P. Liu, L. Li, K. Liu, S. McIntosh, and J. Grundy, "Understanding the quality and evolution of android app build systems," *Journal of Software: Evolution and Process*, vol. 36, no. 5, p. e2602, 2024.

[16] "F-droid - free and open source android app repository," 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://f-droid.org/

[17] "Android ndk — android developers," 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://developer.android.com/ndk

[18] "Cmake - upgrade your software build system," [Online; accessed 2025-03-13]. [Online]. Available: https://cmake.org/

[19] S. P. M. a. G. Jon Janego, "Gradle build tool," 2 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://gradle.org/

[20] "simlar-android/scripts/bootstrap-liblinphone.sh at master · simlar/simlar-android," [Online; accessed 2025-03-14]. [Online]. Available: https://github.com/simlar/simlar-android/blob/master/scripts/bootstrap-liblinphone.sh

[21] "meypod/al-azan: Privacy focused ad-free open-source muslim adhan (islamic prayer times) and qibla app," [Online; accessed 2025-03-14]. [Online]. Available: https://github.com/meypod/al-azan/tree/main

[22] "przemekr / dot-race android / jni / src / android.mk — bitbucket," 12 2014, [Online; accessed 2025-03-14]. [Online]. Available: https://bitbucket.org/przemekr/dot-race/src/master/android/jni/src/Android.mk

[23] droidcon, "Mastering gradle dependency management with version catalogs: A comprehensive guide - droidcon," 4 2023, [Online; accessed 2025-03-13]. [Online]. Available: https://www.droidcon.com/2023/04/12/mastering-gradle-dependency-management-with-version-catalogs-a-comprehensive-guide/

[24] "podverse-fdroid/android/build.gradle at develop · podverse/podverse-fdroid," [Online; accessed 2025-03-14]. [Online]. Available: https://github.com/podverse/podverse-fdroid/blob/develop/android/build.gradle

[25] "podverse-fdroid/android/app/build.gradle at develop · podverse/podverse-fdroid," [Online; accessed 2025-03-14]. [Online]. Available: https://github.com/podverse/podverse-fdroid/blob/develop/android/app/build.gradle

[26] "Imagetoolbox/gradle/libs.versions.toml at master · t8rin/imagetoolbox," [Online; accessed 2025-03-14]. [Online]. Available: https://github.com/T8RIN/ImageToolbox/blob/master/gradle/libs.versions.toml

[27] "Qt — tools for each stage of software development lifecycle," 10 2025, [Online; accessed 2025-10-17]. [Online]. Available: https://www.qt.io/

[28] "Google's maven repository," 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://maven.google.com/web/index.html

[29] "Central repository:," 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://repo.maven.apache.org/maven2/

[30] adia, "Jcenter sunset on august 15th, 2024 — jfrog," 7 2024, [Online; accessed 2025-03-14]. [Online]. Available: https://jfrog.com/blog/jcenter-sunset/

[31] giannit, "Service end for bintray, jcenter, gocenter, and chartcenter — jfrog," 2 2021, [Online; accessed 2025-03-14]. [Online]. Available: https://jfrog.com/blog/into-the-sunset-bintray-jcenter-gocenter-and-chartcenter/

[32] "Index of /ubuntu/pool/main," 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://archive.ubuntu.com/ubuntu/pool/main/

[33] "Index of /debian/pool/main," 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://ftp.debian.org/debian/pool/main/

[34] JitPack.io, "Jitpack — publish jvm and android libraries," 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://jitpack.io/

[35] "Software supply chain management — sonatype," 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://www.sonatype.com/

[36] "Gradle - plugin: com.jfrog.bintray," 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://plugins.gradle.org/plugin/com.jfrog.bintray

[37] "Commonsware," 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://commonsware.com/

[38] "Clojars," 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://clojars.org/

[39] "Assigned to me - issue tracker," 2025, [Online; accessed 2025-03-15]. [Online]. Available: https://issuetracker.google.com

[40] A. Abraham, "Mobile security framework - mobsf," 2025, [Online; accessed 2025-03-15]. [Online]. Available: https://www.mobsf.live/

[41] ReFirmLabs, "Github - refirmlabs/binwalk: Firmware analysis tool," 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://github.com/ReFirmLabs/binwalk/

[42] "Google open source," 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://opensource.google/

[43] "Github · build and ship software on a single, collaborative platform · github," 1 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://github.com/

[44] "Apkcombo," 2025, [Online; accessed 2025-03-03]. [Online]. Available: https://apkcombo.app/

[45] J. M. Gonzalez-Barahona, "Characterizing outdateness with technical lag: an exploratory study," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 735–741.

[46] T. Preston-Werner, "Semantic versioning 2.0.0 — semantic versioning," [Online; accessed 2025-03-15]. [Online]. Available: https://semver.org/

[47] C. to Wikimedia projects, "Pearson correlation coefficient - wikipedia," 5 2003, [Online; accessed 2025-03-15]. [Online]. Available: https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

[48] ——, "Spearman's rank correlation coefficient - wikipedia," 5 2003, [Online; accessed 2025-03-15]. [Online]. Available: https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient

[49] X. Zhang, L. Fan, S. Chen, Y. Su, and B. Li, "Scene-driven exploration and gui modeling for android apps," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '23. IEEE Press, 2024, p. 1251–1262. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00179