Too Quiet in the Library: An Empirical Study of Security Updates in Android Apps' Native Code

Sumaya Almanee*, Arda Ünal*, Mathias Payer†, and Joshua Garcia*

*University of California Irvine, {salmanee, unala, joshug4}@uci.edu
†EPFL, mathias.payer@nebelwelt.net

***Abstract*—Android apps include third-party native libraries to increase performance and to reuse functionality. Native code is directly executed from apps through the Java Native Interface or the Android Native Development Kit. Android developers add precompiled native libraries to their projects, enabling their use. Unfortunately, developers often struggle or simply neglect to update these libraries in a timely manner. This results in the continuous use of outdated native libraries with unpatched security vulnerabilities years after patches became available.**

To further understand such phenomena, we study the security updates in native libraries in the most popular 200 free apps on Google Play from Sept. 2013 to May 2020. A core difficulty we face in this study is the identification of libraries and their versions. Developers often rename or modify libraries, making their identification challenging. We create an approach called *LibRARIAN* (LibRAry veRsion IdentificAtioN) that accurately identifies native libraries and their versions as found in Android apps based on our novel similarity metric bin^2sim. *LibRARIAN* leverages different features extracted from libraries based on their metadata and identifying strings in read-only sections.

We discovered $53/200$ popular apps (26.5%) with vulnerable versions with known CVEs between Sept. 2013 and May 2020, with 14 of those apps remaining vulnerable. We find that app developers took, on average, 528.71 ± 40.20 days to apply security patches, while library developers release a security patch after 54.59 ± 8.12 days—a 10 times slower rate of update.

I. Introduction

Third-party libraries are convenient, reusable, and form an integral part of mobile apps. Developers can save time and effort by reusing already implemented functionality. Native third-party libraries are prevalent in Android applications ("apps"), especially social networking and gaming apps. These two app categories—ranked among the top categories on Google Play—require special functionality such as 3D rendering, or audio/video encoding/decoding [14], [26], [40], [33], [38]. These tasks tend to be resource-intensive and are, thus, often handled by native libraries to improve runtime performance.

The ubiquity of third-party libraries in Android apps increases the attack surface [35], [39] since host apps expose vulnerabilities propagated from these libraries [20], [37]. Another series of previous work has studied the outdatedness and updateability of third-party *Java libraries* in Android apps [9], [4], with a focus on managed code of such apps (e.g., Java or Dalvik code). However, these previous studies do not consider *native libraries* used by Android apps.

We argue that security implications in native libraries are even more critical for three main reasons. First, app developers add native libraries but neglect to update them. The reasons for this may include concerns over regressions arising from such updates, prioritizing new functionality over security, deadline pressures, or lack of tracking library dependencies and their security patches. This negligence results in outdated or vulnerable native libraries remaining in new versions of apps. Second, native libraries are susceptible to memory vulnerabilities (e.g., buffer overflow attacks) that are straight-forward to exploit. Third, and contrary to studies from almost 10 years ago [11], [44], native libraries are now used pervasively in mobile apps. To illustrate this point, we analyzed the top 200 apps from Google Play between Sept. 2013 and May 2020. We obtained the version histories of these apps from *AndroZoo* [2] totaling $7{,}678$ versions of those 200 top free apps. From these apps, we identified $66{,}684$ native libraries in total with an average of 11 libraries per app and a maximum of 141 for one version of *Instagram*.

To better understand the usage of third-party native libraries in Android apps and its security implications, we conduct a longitudinal study to identify vulnerabilities in third-party native libraries and assess the extent to which developers update such libraries of their apps. In order to achieve this, we make the following research contributions:

- We construct a novel approach, called *LibRARIAN* (**LibRA**ry ve**R**sion Identific**A**tio**N**) that, given an unknown binary, identifies (i) the library it implements and (ii) its version. Furthermore, we introduce a new similarity-scoring mechanism for comparing native binaries called bin^2sim, which utilizes 6 features that enable *LibRARIAN* to distinguish between different libraries and their versions. The features cover both metadata and data extracted from the libraries. These features represent elements of a library that are likely to change between major, minor, and patch versions of a native library.
- We conduct a large-scale, longitudinal study that tracks security vulnerabilities in native libraries used in apps over 7 years. We build a repository of Android apps and their native libraries with the 200 most popular free apps from Google Play totaling $7{,}678$ versions gathered between the dates of Sept. 2013 and May 2020. This repository further contains $66{,}684$ native libraries used by these $7{,}678$ versions.

Prior work [30], [24], [21], [1] has measured the similarity

between binaries. However, these approaches identify semantic similarities/differences between binaries at the *function-level*, with the goal of identifying malware. *LibRARIAN*, orthogonally, is a syntactic-based tool which computes similarity between two benign binaries (at the file-level) with the goal of identifying library versions with high scalability.

We utilize *LibRARIAN* and our repository to study (1) *LibRARIAN*'s accuracy and effectiveness, (2) the prevalence of vulnerabilities in native libraries in the top 200 apps, and (3) the rate at which app developers apply patches to address vulnerabilities in native binaries. The major findings of our study are as follows:

- For our ground truth dataset which contains 46 known libraries with 904 versions, *LibRARIAN* correctly identifies 91.15% of those library versions, thus achieving a high identification accuracy.
- To study the prevalence of vulnerabilities in the top 200 apps in Google Play, we use *LibRARIAN* to examine 53 apps with vulnerable versions and known CVEs between Sept. 2013 and May 2020. 14 of these apps remain vulnerable and contain a wide-range of vulnerability types—including denial of service, memory leaks, null pointer dereferences, or divide-by-zero errors. We further find that libraries in these apps, on average, have been outdated for 859.17 ± 137.55 days. The combination of high severity and long exposure of these vulnerabilities results in ample opportunity for attackers to target these highly popular apps.
- To determine developer response rate of applying security fixes, we utilize *LibRARIAN* to analyze 40 apps, focusing on popular third-party libraries (those found in more apps) with known CVEs such as *FFmpeg*, *GIFLib*, *OpenSSL*, *WebP*, *SQLite3*, *OpenCV*, *Jpeg-turbo*, *Libpng*, and *XML2*, between Sept. 2013 and May 2020.
  We find that app developers took, on average, 528.71 ± 40.20 days to apply security patches, while library developers release a security patch after 54.59 ± 8.12 days—a 10 times slower rate of update. These libraries that tend to go for long periods without being patched affect highly popular apps with billions of downloads and installs.
- We make our dataset, analysis platform, and results available online to enable reusability, reproducibility, and others to build upon our work [25].

II. *LibRARIAN*

Figure 1 shows the overall workflow of *LibRARIAN*. *LibRARIAN* identifies unknown third-party native libraries and their versions (*Unknown Lib Versions*) by (1) extracting features that distinguish major, minor, and patch versions of libraries that are stable across platforms regardless of underlying architecture or compilation environments; (2) comparing those features against features from a ground-truth dataset (*Known Lib Versions*) using a novel similarity metric, bin^2sim; and (3) matching against strings that identify version information of libraries extracted from *Known Lib Versions*, which we refer to as *Version Identification Strings*. In the remainder of this section, we describe each of these three major steps of *LibRARIAN*.

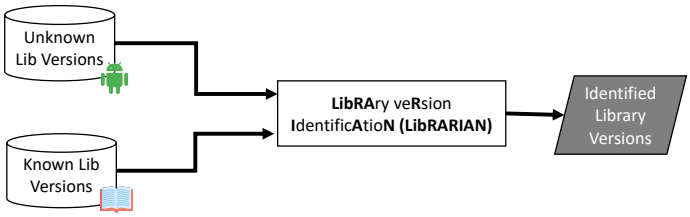

Fig. 1: *LibRARIAN* identifies versions of native binaries from Android apps by using our bin^2sim similarity-scoring technique to compare known (ground-truth dataset) and unknown versions of native binaries.

A. Feature Vector Extraction

Our binary similarity detection is based on the extraction of features from binaries combining both metadata found in Executable and Linkable Format (ELF) files as well as identifying features in different binary sections of the library. All shared libraries included in Android apps are compiled into ELF binaries. Like other object files, ELF binaries contain a symbol table with externally visible identifiers such as function names, global symbols, local symbols, and imported symbols. This symbol table is used (1) during loading and linking and (2) by binary analysis tools [16] (e.g., `objdump`, `readelf`, `nm`, `pwntools`, or `angr` [36]) to infer information about the binary.

To distinguish between different libraries and their versions, we need to identify *differencing features*. To that end, we define a set of six features inherent to versions and libraries. Five features represent ELF metadata, these features are used to compute the similarity score between two binaries as described in Section II-B, hence, we refer to these features as *Metadata Features*. Orthogonally, we leverage strings extracted from the *.rodata* section of an ELF object, which we refer to as *Version Identification Strings*. This feature complements the similarity score from the first set of features. We either use it to verify the correctness of the version or as a fallback if the similarity to existing binaries in our ground-truth dataset is low (see Section II-C).

Table I shows the list of all *LibRARIAN* features. The features include: (i) five *Metadata Features* based on exported and imported functions, exported and imported globals, and library dependencies; and (ii) one *Data Feature* which is applied as a second factor to either substitute the *Metadata Features*, in case the reported similarity score is low, or to confirm the reported score. These 6 features represent the code elements of a library that would be expected to change based on a versioning scheme that distinguishes major, minor, and patch versions of a library. Furthermore, these features are stable across platforms regardless of the underlying architecture or compilation environments. We did not include code features (e.g., control-flow and data-flow features) as they are extremely volatile and change between compilations and across architectures. Binary similarity matching is a hard open

Feature Type	Name	Definition
Metadata	Exported Globals	Externally visible variables, i.e., they can be accessed externally.
	Imported Globals	Variables from other libraries that are used in this library.
	Exported Functions	Externally visible functions, i.e., functions that can be called from outside the library.
	Imported Functions	Functions from other libraries that are used in this library.
	Dependencies	The library dependencies that are automatically loaded by the ELF object
Data	Version Identification Strings	Flexible per-library version strings (e.g., "libFoo-1.0.2" matched to strings in the *.rodata* section of an ELF object)

TABLE I: List of features *LibRARIAN* extracts from native binaries of Android apps along with their type and definition.

problem: While recent work has made progress regarding accuracy [12], [42], [30], [24], [21], [1], [19], [10], the majority of algorithms have exponential computation cost relative to the code size and are infeasible for large-scale studies.

We built a dataset of heuristics by inspecting the binaries in our ground-truth dataset. We developed scripts to process the data in the *.rodata* sections extracted during feature processing and search for unique per-library strings that contain version information. For example, *FFmpeg* version info is found when applying the regex `ffmpeg-([0-9]\.)*[0-9]` or `FFmpeg version([0-9]\.)*[0-9]`. Table II shows our list of extracted version heuristics. Each version heuristic can be produced automatically by constructing regular expressions from strings in *.rodata* sections of binaries in our ground-truth dataset. For example, if the string "*libFoo-1.0.2*" is found in version 1.0.2 of *libFoo*, *LibRARIAN* uses a regular expression replacing the numeric suffix of the string with an appropriate pattern (e.g., `libFoo-[0-9]+(\.[0-9])*`).

We deliberately exclude any metadata or identifying strings for symbols that are volatile across architectures or build environments like compiler version, relocation information (and types), or debug symbols. *LibRARIAN*'s accuracy results in Section III-A2 demonstrate that our selected set of features suffice to distinguish between different versions of libraries.

The implementation leverages angr's [36] ELF parser which already is platform independent. Our extraction platform recovers all metadata from the ELF symbol tables and, if available, searches for string patterns in the comment and read-only sections. Our filters remove platform specific information and calls to standard libraries (e.g., C++ ABI calls, vectors, or other data structures). The current implementation covers x86-64, x86, ARM, and ARM64 binaries—which are all platforms we observed in our evaluation. We accommodate for architecture differences in two ways: First, we remove architecture noise in feature vectors (e.g., symbols that are only used in one architecture); and second, we collect, if available, binaries for the different architectures.

The feature extraction compiles all recovered information as a dictionary into a JSON file. The dictionary contains arrays of strings for each of the features mentioned above plus additional metadata to identify the library and architecture.

B. Similarity Computation

LibRARIAN's similarity computation, which we refer to as bin^2sim, leverages the five *Metadata Features* when computing the similarity scores between an app binary and our ground-truth dataset. bin^2sim is based on the *Jaccard coefficient*, and is used to determine the similarity between feature vectors. bin^2sim allows *LibRARIAN* to account for addition or removal of features between different libraries and versions. Given two binaries b_1 and b_2 with respective feature vectors FV_1 and FV_2, bin^2sim computes the size of the intersection of FV_1 and FV_2 (i.e., the number of common features) over the size of the union of FV_1 and FV_2 (i.e., the number of unique features):

$$bin^2sim(FV_1, FV_2) = \frac{\mid FV_1 \cap FV_2 \mid}{\mid FV_1 \cup FV_2 \mid} \in [0, 1] \quad (1)$$

The similarity score is a real number between 0 and 1, with a score of 1 indicating identical features, a score of 0 indicating no shared features between the two libraries, and a fractional value indicating a partial match. Due to the volatility of the similarity score, filtering noise such as platform-specific details as mentioned in the previous section is essential for the accuracy of our approach.

LibRARIAN counts an unknown library instance from *Unknown Lib Versions* as matching a known library version if its bin^2sim is above 0.85. This threshold was determined experimentally and works effectively as our evaluation will demonstrate (see Section III). If bin^2sim results in the same value above the threshold for multiple known binaries, *LibRARIAN* tries obtaining an exact match between one of the known binaries and the unknown binary by using their hash codes to determine the unknown binary's version.

A low similarity score might result from modifications made by app developers to the original third-party library which results in the removal or addition of specific features. From our experience, removal of features from the original library is common among mobile developers and is likely driven by the need to reduce the size of the library and the app as much as possible. For example, we observed that the WebP video codec library is often deployed without encoding functionalities to reduce binary size. Some size optimization techniques require choosing needed modules from a library and leaving the rest, stripping the resulting binary, and modifying build flags. Another factor that reduces similarity as measured by the Jaccard coefficient is that certain architectures tend to export more features as compared to others. For instance, 32-bit architectures such as armeabi-v7a and x86 export more features compared to arm64-v8a and x86_64.

C. Version Identification Strings

For libraries where *LibRARIAN* reports low similarity scores (e.g., some libraries like *RenderScript* or *Unity* only

Library Name	Extracted Heuristics
Jpeg-turbo	`Jpeg-turbo version 1(\.[0-9]{1,})*`
FFmpeg	`ffmpeg-([0-9]\.)*[0-9]|FFmpeg version ([0-9]\.)*[0-9]`
Firebase	`Firebase C++ [0-9]+(\.[0-9])*`
Libavcodec	`Lavc5[0-9](\.[0-9]{1,})`
Libavfilter	`Lavf5[0-9](\.[0-9]{1,})`
Libpng	`Libpng version 1(\.[0-9]{1,})*`
Libglog	`glog-[0-9]+(\.[0-9])*`
Libvpx	`WebM Project VP(.*)`
OpenCV	`General configuration for OpenCV [0-9]+(\.[0-9])*|opencv-[0-9]+(\.[0-9])*`
OpenSSL	`openssl-1(\.[0-9])*[a-z]|^OpenSSL 1(\.[0-9])*[a-z]`
Speex	`speex-(.*)`
SQLite3	`^3\.([0-9]{1,}\.)+[0-9]`
Unity3D	`([0-9]+\.)+([0-9]+)[a-z][0-9]|Expected version:(.*)`
Vorbis	`Xiph.Org Vorbis 1.(.*)`
XML2	`GITv2.[0-9]+(\.[0-9])`

TABLE II: Heuristics used to search for unique per-library strings that contain version information

export a single function [1]), these five features fail to provide sufficient information about the underlying components in a library. If libraries only export one or a few functions, the similarity metrics have a hard time distinguishing between different libraries. We therefore extend the features with strings that uniquely identify the library. Such strings are often version strings. Based on extracted flexible per-library heuristics from our ground-truth dataset (see Table II), we heuristically identify exact library versions and increase overall accuracy. For libraries with high similarity scores, we use these library heuristics to confirm the correct version.

To identify binaries with low similarity scores, we leverage *Version Identification Strings*, which is the set of extracted per-library version strings. For example, say a library version lv extracted from app a had a similarity score of 0.3 when compared with *OpenCV-2.4.11* using *Metadata Features*. Given the low score, we search the *Version Identification Strings* feature for specific keywords such as `General configuration for OpenCV *.*.*` or `opencv-*.*.*`. Where the asterisk represents the versioning scheme of *OpenCV* library.

Our feature extraction process logs all strings (arrays of more than 3 ASCII printable characters ending with a 0 byte) from the *.rodata* section alongside the other features. As libraries commonly have large amounts of read-only string data that frequently changes, we cannot use this data directly as a feature (due to the low overlap resulting in low similarity). By processing the *.rodata* from our ground-truth dataset and clustering the data, we extract common version identifiers and version strings. We then translate them into regular expressions that allow us to match versions for different libraries.

III. Evaluation

To assess the prevalence of vulnerable native libraries for Android, we answer the following three research questions:

RQ1: *Accuracy and effectiveness of LibRARIAN.* Can *LibRARIAN* accurately and effectively identify versions of native libraries? How does *LibRARIAN* compare against state-of-the-art native-library version identification? How effective are *LibRARIAN*'s feature types at identifying versions of native libraries?

RQ2: *Prevalence of outdated libraries.* How prevalent are vulnerabilities in native libraries of Android apps?

RQ3: *Patch response time.* After a vulnerability is reported for a third-party library, how quickly do developers apply patches?

To supplement the aforementioned RQs, we conducted a detailed case study on a vulnerable app (Section III-D), providing practical insight into vulnerabilities in third-party libraries and possible exploits.

To answer these research questions, we analyze the top 200 apps in Google Play over several years. We track the version history of these apps from AndroZoo [2], a large repository of over 11 million Android apps. Our repository contains app metadata including the app name, release dates, and native binaries. Note that Google Play unfortunately restricts lists to 200 apps. Overall, we collected 7,678 instances, where each instance is a version of the 200 top apps from Google Play.

We determined that 145 out of 200 (72.50%) of the distinct apps in our repository contain at least one native library, i.e., 5,852 out of 7,678 (76.21%) of the total apps in our database. There are a total of 66,684 libraries in the form of *.so* files, i.e., shared library files, in our repository with an average of 11 libraries per app and a maximum of 141 for one version of *Instagram*. In fact, *Instagram*—for which we collected 184 versions since Dec. 2013—contains a total of 6,677 *.so* files.

We run *LibRARIAN* on a machine with 2 AMD EPYC 7551 32-Core CPUs and 512GB of RAM running Ubuntu 18.04. The average number of features in the extracted feature vectors is 2,116.86 features. Some outliers such as *libWaze* and *libTensorflow* reach up to 79,581 features. This shows that the set of third-party native libraries in our repository is diverse, some of them are very complex and offer a large number of functionalities. Generating feature vectors is quick and generally takes a few seconds per library. The most complex library, *libTensorflow* takes 4 min and 38 sec to analyze. We found that, out of 7,253 binaries for which *LibRARIAN* inferred their versions, the average runtime for library version detection is 118.19 seconds—with a minimum of 97 seconds and a maximum of 224 seconds.

[1]These libraries are "stripped" and hide all functionality internally. The single exported function takes a string as parameter which corresponds to the target function and they dispatch to internal functionality based on this string.

A. RQ1: Accuracy and Effectiveness

To determine if *LibRARIAN* accurately and effectively identifies native library versions from Android apps, we assess *LibRARIAN* in three scenarios. For the first scenario, we compare its accuracy with *OSSPolice*, the state-of-the-art technique for identifying versions of native binaries for Android apps. For the second scenario, we assess *LibRARIAN* on a larger dataset for which *OSSPolice* could not be applied and, thus, evaluate *LibRARIAN*'s accuracy independently of other tools. In the third scenario, we assess the effectiveness of *LibRARIAN*'s feature types at identifying versions of native libraries.

1) Comparative Analysis: *OSSPolice* uses source code to build an index that allows it to identify versions of binaries. *OSSPolice* measures the similarity between strings extracted from binaries and features found directly in source repositories. Unlike *LibRARIAN*, *OSSPolice* relies on comparing binaries with source code, resulting in an overly large feature space which, in turn, makes *OSSPolice* susceptible to falsely identifying any binary containing a library as exactly matching that library. For example, *OSSPolice* falsely identifies *MuPDF* and *OpenCV* as matching *Libpng* because those two libraries include *Libpng* in their source code [10].

We repeatedly contacted the *OSSPolice* authors to obtain a fully-working version of their tool, but unfortunately they did not provide us their non-public data index or sufficient information to reproduce their setup. As a result, we performed a comparative analysis between *LibRARIAN* and *OSSPolice* based on the published *OSSPolice* numbers [10].

The ground-truth dataset in the *OSSPolice* evaluation contains a total of 475 binaries (out of which 67 are unique) extracted from 104 applications collected by F-Droid [13]. *LibRARIAN* correctly identified $63/67$ (94%) unique binaries in the *OSSPolice* dataset, improving accuracy by 12% compared to the accuracy reported by *OSSPolice* (82%) which correctly identified $55/67$ libraries. *OSSPolice* has lower accuracy because it misidentifies reused libraries (as described above) and it relies on simple syntactic features (e.g., string literals and exported functions) while our feature vectors extract additional features—such as imported functions, exported and imported global variables, and dependencies that uniquely identify different versions of binaries. These additional features were a major factor in the superior accuracy of *LibRARIAN* compared to *OSSPolice*.

LibRARIAN did not identify 4 binaries because the library functions are dispatched from a single function and do not contain identifying version information that was readily available. Hence, our extracted features fail to provide sufficient information about the underlying components in the library. Nevertheless, *LibRARIAN* significantly reduces the number of binaries that need to be manually inspected.

Lastly, it is important to reiterate that these results are only compared against the dataset used in the *OSSPolice* paper but without us being able to replicate or reuse *OSSPolice*, due to key elements of the tool being unavailable.

Finding 1: *LibRARIAN* achieves a 12% improvement in its accuracy compared to *OSSPolice* on the 67 unique binaries in *OSSPolice*'s dataset. Unlike *OSSPolice*, *LibRARIAN* obtains this improvement without relying on source code, which may not be available for all libraries and results in an unnecessarily larger feature space.

2) Independent Accuracy: We further assess *LibRARIAN*'s accuracy on a larger and more recent set of library versions than those found in *OSSPolice*'s dataset. To that end, we manually collect a set of binaries with known libraries and versions (*Known Lib Versions* in Figure 1) and compare the inferred libraries and versions to the known ones to determine *LibRARIAN*'s accuracy. We build our dataset based on libraries used in common Android apps.

Experiment Setup. We first manually locate the pre-built binaries of libraries to serve as ground truth. To that end, we use readily available auxiliary data such as keywords found in feature vectors, binary filenames, and dependencies. Once we identify potential targets, we retrieve the pre-built binaries of all versions and architectures, if possible.

There are a variety of distribution channels where app developers can obtain third-party binaries. We obtained such binaries from official websites, GitHub, and Debian repositories. The binaries with known libraries and versions contain 46 distinct libraries with a total of 904 versions and an average of 19 versions per library.

Results. *LibRARIAN* correctly identified the versions of $824/904$ (91.15%) libraries in our ground truth: $553/904$ (61.17%) of these library versions have unique feature vectors; 15.16% of the these libraries contain the exact version number in the strings literals; and the remaining 14.82% of library versions are distinguished using hash codes to break ties between bin^2sim values of binaries.

Misidentification occurs in 8.85% of library versions, where the largest equivalence class contains 4 library versions. This usually occurs for consecutive versions—minor or micro revisions (e.g., 3.1.0 and 3.1.1). These minor or micro revisions generally fix small bugs and do not change, add, or remove exported symbols. Although *LibRARIAN* cannot pinpoint the exact library version in this case, *LibRARIAN* significantly reduces the search space for post analysis to a few candidate versions.

Finding 2: *LibRARIAN* correctly identifies 824 of 904 (91.15%) library versions from 46 distinct libraries, making it highly accurate for identifying the native libraries and versions. For misidentified library versions, *LibRARIAN* reports a slightly different version.

3) Feature Effectiveness: To assess the effectiveness of *Metadata Features*, *Version Identification Strings*, and their combination at inferring binaries, we computed the extent to which each feature is capable of inferring binaries in our repository. To that end, any binary whose library and version can be inferred with a bin^2sim above 0.85 as described

in Section II counts as an inferred binary. We found that 37.42% of binaries in our repository are inferable by *Version Identification Strings* only. 45.29% of the remaining binaries are inferable using only the five *Metadata Features* mentioned in Section II-A, while the remaining 17.29% are inferred using both *Metadata Features* and *Version Identification Strings*. This indicates that not all libraries have the version information encoded directly in the strings. Having a combination of both *Metadata Features* and *Version Identification Strings* is crucial to increase the number of inferred binaries.

We further aimed to assess the extent to which each of the five *Metadata Features* contribute to computing bin^2sim in order to assess each of their individual effectiveness. Recall from Section II-B that our matching algorithm leverages five features when computing the similarity scores between an app binary and our ground-truth dataset. Table III lists these feature along with their contribution factor, i.e., the average percentage each one of these features contribute to the total similarity score. To calculate the contribution factor ($contrib_f$) of a feature f, we first calculate the similarity score taking all five features into account ($score_{all}$). We then calculate the similarity score of each feature separately ($score_f$). For each feature, we find $contrib_f = score_f / score_{all}$, which is the percentage each f contributes to the total similarity score. As shown in Table III, *Exported Functions* contributes the most when computing bin^2sim (Equation 1), i.e., 58.25% of the matching features are *Exported Functions*, followed by *Imported Functions* contributing 32.98%, *Dependencies*, *Exported Globals*, and finally *Imported Globals* contributing less overall. Still, these five features sometimes manage to uniquely identify a library and are therefore included as they, overall, improve the similarity score. Recall that *Version Identification Strings* is not taken into account when computing the similarity score between binaries.

Finding 3: 37.42% of binaries are inferable using *Version Identification Strings*, 45.29% are inferable using *Metadata Features*, and *17.29%* are inferable using both feature types. *Exported Functions* and *Imported Functions* account for the overwhelming majority of effectiveness of *Metadata Features*, contributing 58.25% and 32.98%, respectively.

Feature Type	Name	Contribution Factor
Metadata	Exported Globals	3.32%
	Imported Globals	1.06%
	Exported Functions	58.25%
	Imported Functions	32.98%
	Dependencies	4.39%

TABLE III: List of features bin^2sim extracted from native binaries of Android apps along with their type and overall contribution factor, which measures the average percentage each feature contributes to the total similarity score

B. RQ2: Prevalence of Vulnerable Libraries

To study the prevalence of vulnerabilities in native libraries, we need to identify their exact versions. To that end, we leverage *LibRARIAN* to identify potential library versions from our repository. Once the versions are identified, we investigate the extent to which native libraries of Android apps are vulnerable and remain vulnerable.

Experiment Setup. We infer the correct version of $7{,}253$ binaries (10.87% of the total binaries in our Android repository) using *LibRARIAN*. Due to the highly time-consuming nature of the manual collection of ground-truth binaries, we limit ourselves to libraries that (i) are found in a greater number of apps (more than 10 apps) and (ii) have known *CVEs*. As a result, an overwhelming majority of the remaining binaries in our dataset have either no known CVEs or affect very few apps, making them an unsuitable choice for applying an expensive manual analysis for studying this research question.

Results. We found that, out of $7{,}253$ binaries for which we inferred their versions, $3{,}674$ were vulnerable libraries (50.65%) affecting $53/200$ distinct apps. 14 new releases of these distinct apps remain vulnerable at the time of submission. The complete list of libraries with reported *CVEs* between Sept. 2013 and the writing of this paper can be found in Table IV. As for the number of apps affected by vulnerable libraries, our results show that 53 distinct apps have been affected by a minimum of 1 vulnerable library and a maximum of 16 vulnerable libraries covering dates between Sept 2013 and May 2020.

Finding 4: 53 of the 200 top apps on Google Play (26.5%) were plagued by a vulnerable library over approximately six years and 8 months (i.e., between Sept. 2013 and May 2020). 14 of those apps still include a vulnerable binary, i.e., 7% of the top 200 apps on Google Play, even at the time at which we collected apps for this study and are, on average, outdated by 859.17 ± 137.55 days. As a result, vulnerable native libraries play a substantial role in exposing popular Android apps to known vulnerabilities.

We emailed app developers since February 2020 to inform them that their apps continue to use a vulnerable library. We urged them to take an action (i.e., remove or replace such libraries) or at least provide some justification as to why such libraries are not updated. Our investigation is ongoing. While several app developers already updated their apps to remove the vulnerable library, many updates are still outstanding. Some of the replies we received simply blame other library developers. For example, we heard back from Discord that the vulnerable lib is a dependency of another third-party library used in Discord (Fresco): "Until Fresco fixes this, however, we are not able to address this in our app".

Four libraries were particularly prevalent in terms of the number of vulnerable versions they contain (i.e., *OpenSSL*), the number of apps they affect (i.e., *OpenCV* and *GIFLib*), or the length of time during which the library remained vulnerable (i.e., *XML2* in *Microsoft XBox SmartGlass*). *OpenSSL* has the largest number of vulnerable versions (22 in total) included in 13 distinct apps. 3 apps: Amazon Alexa, Facebook Messenger and Norton Secure VPN still include vulnerable

Lib Name	No. Vul Lib Vers	Vul Lib Vers	No. Apps	No. Apps Still Vul
OpenCV	5	2.4.1, 2.4.11, 2.4.13, 3.1.0, 3.4.1	21	7
WebP	3	0.3.1, 0.4.2, 0.4.3	11	1
GIFLib	2	5.1.1, 5.1.4	15	1
FFmpeg	9	2.8, 2.8.7, 3.0.1, 3.0.3, 3.2, 3.3.2, 3.3.4, 3.4, 4.0.2	8	1
Libavcodec	9	55.39.101, 55.52.102, 56.1.100, 56.60.100, 57.107.100, 57.17.100, 57.24.102, 57.64.100, 57.89.100	10	0
Libavformat	3	55.19.104, 56.40.101, 57.71.100	3	0
Libavfilter	3	3.90.100, 4.2.100, 5.1.100	1	0
Libavutil	3	52.48.101, 52.66.100, 54.20.100	2	0
Libswscale	3	2.5.101, 3.0.100, 4.0.100	5	1
Libswresample	2	0.17.104, 1.1.100	1	0
SQlite3	7	3.11.0, 3.15.2, 3.20.1, 3.26.0, 3.27.2, 3.28.0, 3.8.10.2	7	2
XML2	1	2.7.7	3	1
OpenSSL	22	1.0.0a, 1.0.1c, 1.0.1e, 1.0.1h, 1.0.1i, 1.0.1p, 1.0.1s, 1.0.2a, 1.0.2f, 1.0.2g, 1.0.2h, 1.0.2j, 1.0.2k, 1.0.2m, 1.0.2o, 1.0.2p, 1.0.2r, 1.1.0, 1.1.0g, 1.1.0h, 1.1.0i, 1.1.1b	13	3
Jpeg-turbo	2	1.5.1, 1.5.2	3	0
Libpng	7	1.6.10, 1.6.17, 1.6.24, 1.6.34, 1.6.37, 1.6.7, 1.6.8	5	1

TABLE IV: A list of libraries with reported *CVEs* found in our repository along with the number of distinct apps that were affected by a vulnerable library and the number of distinct apps containing a vulnerable version till now.

versions of *OpenSSL*.

OpenCV and *GIFLib* affect the most apps. *OpenCV* has the largest number of affected apps with a total of 21 apps where 7 recent apps still have a vulnerable instance of *OpenCV*. Most applications do not include OpenCV directly but indirectly through the dependencies of `card.io` which enables card payment processing but comes with the two outdated versions (2.4.11 and 2.4.13) of both `opencv_core` and `opencv_imgproc`. Following *OpenCV* in the number of affected apps is *GIFLib*, which has two vulnerable versions found in a total of 15 distinct apps, 1 app is still affected.

One vulnerable version of *XML2* (2.7.7) was found in 35 versions of *Microsoft XBox SmartGlass* and the library was not updated for 6 years—still remaining vulnerable up to the writing of this paper. This particular case is notable due to the extremely long amount of time the library had been vulnerable and remained vulnerable.

To examine the affects of vulnerable libraries on apps further, we list popular apps and the reported CVEs they expose their users to. Table V shows 10 out of 14 popular apps that are using at least one library with a reported CVE at the time of our app collection. We discuss four of these apps in more detail in the remainder of this section.

Facebook Messenger, which has a download base of over 500M (the largest in this list), contains *OpenSSL-1.1.0*, which is vulnerable since Sept. 2016. This vulnerable library contains multiple memory leaks which allows an attacker to cause a denial of service (memory consumption) by sending large OCSP (Online Certificate Status Protocol) request extensions.

Amazon Kindle, an app that provides access to an electronic library of books—with a total of more than 100M installs, uses two vulnerable libraries: *XML2-2.7.7* and *Libpng-1.6.7*. *XML2-2.7.7* contains a variant of the "billion laughs" vulnerability which allows attackers to craft an XML document with a large number of nested entries that results in a denial of service attack. *XML2-2.7.7* is vulnerable since Nov. 2014 and continues to be used in recent versions of the app. *Libpng-1.6.7* has a NULL pointer dereference vulnerability. This vulnerability was published 6 years ago under *CVE-2013-6954* and it remains unchanged in recent releases of *Amazon Kindle*.

DoorDash, a food delivery app with more than 10M installs includes *GIFLib*-5.1.4 which was reported vulnerable over 8 months ago. A malformed GIF file triggers a division-by-zero exception in the *DGifSlurp* function in *GIFLib* versions prior to 5.1.6. This vulnerable library remains unchanged up to now.

Target uses *OpenCV-2.4.11* as a dependency of `card.io` which enables card payment processing. This version of *OpenCV* was announced vulnerable in Aug. 2017 yet remains unchanged in these apps.

App Name	Vulnerable Libs	No. Installs
Amazon Alexa	OpenSSL-1.0.2p, SQlite3-3.27.2	10M+
Amazon Kindle	Libpng-1.6.7, XML2-2.7.7	100M+
Amazon Music	FFmpeg-4.0.2	100M+
DoorDash	GIFLib-5.1.4	10M+
Facebook Messenger	OpenSSL-1.1.0	500M+
Grubhub	OpenCV-2.4.1	10M+
Sam's Club	OpenCV-2.4.1	10M+
SUBWAY	OpenCV-2.4.1	5M+
Norton Secure VPN	OpenSSL-1.1.1b	10M+
Target	OpenCV-2.4.11	10M+

TABLE V: 10 out of 14 popular apps from Google Play which include a vulnerable library that remained unchanged.

Finding 5: These four apps showcase that these vulnerabilities are wide-ranging involving denial of service, memory leaks, or null pointer dereferences. The high severity and long exposure time of these vulnerabilities results in ample opportunity for attackers to target these highly popular apps.

C. RQ3: Rate of Vulnerable Library Fixing

To determine the vulnerability response rate, we identify the duration between (1) the release time of a security update and (2) the time at which app developers applied a fix either by (i) updating to a new library version or (ii) completely removing a vulnerable library. Recall that we collected the previous

versions of the top 200 apps from Google Play. Moreover, we inferred the library versions from 7,253 libraries using *LibRARIAN*. Given the histories of apps and inferred library versions we can track the library *life span* per app—i.e., the time at which a library is added to an app and when it is either removed or updated to a new version in the app.

To this end, we analyzed 40 popular apps with known vulnerable versions of *FFmpeg*, *GIFLib*, *OpenSSL*, *WebP*, *SQLite3*, *OpenCV*, *Jpeg-turbo*, *Libpng*, and *XML2*, between Sept. 2013 and May 2020. We exclude apps that removed a library before a *CVE* was associated with it and apps containing libraries that are vulnerable up to the writing of this paper. We obtained the date at which a library vulnerability was found; when a security patch was made available for the library; and the time at which either the library was updated to a new version or removed. Table VI shows all the combinations of apps and vulnerable libraries.

Finding 6: On average, library developers release a security patch after 54.59 ± 8.12 days from a reported *CVE*. App developers apply these patches, on average, after 528.71 ± 40.20 days from the date an update was made available—which is about 10 times slower than the rate at which library developers release security patches.

Finding 6 reveals that many popular Android apps expose end-users to long vulnerability periods, especially considering that library developers released fixed versions much sooner. This extreme lag between release of a security patch for a library and the time at which an app developer updates to the patched libraries, or just eliminates the library, indicates that, at best, it is (1) highly challenging for developers to update these kinds of libraries or, less charitably, (2) app developers are highly negligent of such libraries.

App Name	Vul Lib Version	Vul Announced	TTRP (Days)	TTAF (Days)
Xbox	XML2-2.7.7	2014-11-04	12	1956
Apple Music	XML2-2.7.7	2014-11-04	12	1704
TikTok	GIFLib-5.1.1	2015-12-21	87	1429
Zoom Meetings	OpenSSL-1.0.0a	2010-08-17	91	1323
Amazon Alexa	OpenSSL-1.0.1s	2016-05-04	12	1086
Amazon Kindle	Libpng-1.6.34	2017-01-30	330	1019
StarMaker	FFmpeg-3.2	2016-12-23	4	1001
eBay	OpenCV-2.4.13	2017-08-06	41	905
Fitbit	SQlite3-3.20.1	2017-10-12	12	902
Uber	OpenCV-2.4.13	2017-08-06	41	830
Snapchat	SQlite3-3.20.1	2017-10-12	12	670
Discord	GIFLib-5.1.1	2015-12-21	87	665
Lyft	OpenCV-2.4.11	2017-08-06	41	662
Twitter	GIFLib-5.1.1	2015-12-21	87	457
Instagram	FFmpeg-2.8.0	2017-01-23	2	267

TABLE VI: Combinations of 15 apps and particular vulnerable library versions they have contained, the date the vulnerability was publicly disclosed (*Vul announced*), the period between vulnerability disclosure and patch availability in days (i.e. Time-to-Release-Patch (*TTRP*)), and the total number of days elapsed before a fix was made (i.e. Time-to-Apply-Fix (*TTAF*))

Developers applied security patches for vulnerable libraries at a rate as slow as 5.4 years, in the case of *Xbox*, and as fast as 267 days for *Instagram*, where a vulnerable version of *FFmpeg* was removed in that amount of time. In order to determine what type of fix was applied by a developer, we checked the next app version where a vulnerable library was last seen. We found that developers either kept the library but updated to a new version, removed a vulnerable version, or removed all native libraries in an app. In the next paragraphs, we discuss five popular native libraries used in Android apps that exhibit particularly slow fix rates: *FFmpeg*, *OpenSSL*, *GIFLib*, *OpenCV*, and *SQLite3*.

Multiple vulnerabilities were found in versions 2.8 and 3.2 of *FFmpeg* in Dec. 2016 and Jan. 2017, respectively. The number of days a security patch was released for these vulnerable library versions is 4 and 2 days, respectively. However, developers took 267 days to address vulnerabilities in *Instagram*, and nearly 3 years to apply a fix in *Starmaker*.

OpenSSL-1.0.0a and *OpenSSL-1.0.1s* were associated with *CVE-2010-2939* and *CVE-2016-2105* in Aug. 2010, and May 2016 of which *OpenSSL* developers provided a security patch 91 and 12 days after. However, developers of *Zoom* took 1,323 days to apply a fix, while developers of *Amazon Alexa* took 1,086 days.

A heap-based buffer overflow was reported in *GIFLib*-5.1.1 at the end of 2015. The results show that 3 apps using this vulnerable version of *GIFLib* have an average time-to-fix, i.e., total number of days elapsed before a fix was applied, of 850.33 days (2.3 years), which is 10 times slower. This lag time is particularly concerning since *GIFLib* released a fix 87 days after the vulnerable version.

A fix to an out-of-bounds read error that was affecting *OpenCV* through version 3.3 was released 41 days after the CVE was published. The vulnerable versions of this library affects 3 apps in total: *Uber*, *Lyft*, and *eBay*. *OpenCV* has an average time-to-fix of 799 days (i.e., 2 years), which is 19 times slower than the rate at which library developers of *OpenCV* release security patches.

SQLite3 released version 3.26.0, which fixes an integer overflow found in all versions prior to 3.25.3. *Snapchat* and *Fitbit* removed a vulnerable version of *SQLite-3.20.1* library 786 days later.

Finding 7: The results for these five popular native libraries in Android apps show that it often takes years for app developers to update to new library versions—even if the existing version contains severe security or privacy vulnerabilities—placing millions of users at major risk.

To further understand the consequences of outdated vulnerable libraries, we calculated the average time-to-fix across all vulnerable libraries per app. Table VII lists the top 10 apps with the most number of days a vulnerable library remained in an app until a fix for the vulnerability was applied. *Apple Music* had the longest lag between the vulnerable library being introduced and fixed, i.e., 4.66 years. *Uber* was the fastest

App Name	Time-to-Apply-Fix (Days)	No. Installs
Apple Music	1704.00	50M+
Amazon Kindle	1019.00	100M+
eBay	905.00	100M+
Fitbit	902.00	10M+
Snapchat	844.00	1,000M+
Xbox	763.67	50M+
ZOOM Meetings	668.33	100M+
Lyft	662.00	10M+
Amazon Alexa	605.50	10M+
Uber	588.50	500M+

TABLE VII: Top 10 most negligent apps in terms of the average time to fix a vulnerable library

at almost 589 days. Individual apps had as few as over 10 million installs and as many as over a billion installs. Among the social-media apps, *Snapchat*, which has over 1 billion downloads and the largest number of installs among the top 10 apps in Table VII, fixed its vulnerable libraries after 844 days. These very long times to fix vulnerable libraries in highly popular social-media apps places billions of users at high security risk.

Finding 8: The most neglected apps in terms of time to fix vulnerable native libraries range from 588.50 days to nearly five years, affecting billions of users and leaving them at substantial risk of having those libraries exploited. This finding emphasizes the need for future research to provide developers with mechanisms for speeding up this very slow fix rate.

Table VIII lists the top 10 most neglected vulnerable libraries across all apps. *XML2* is the most neglected library with an average time-to-fix of 5 years; *WebP* is the least neglected library with an average time-to-fix of 213.40 days. Among these 10 libraries, the fact that it takes app developers 431.81 days, on average, to update vulnerable versions of *OpenSSL* is particularly concerning due to its security-critical nature.

Lib Name	Time-to-Apply-Fix (Days)	Genre
XML2	1830.00	XML parser
Libpng	923.20	Codec
Jpeg-turbo	841.67	Codec
FFmpeg	720.90	Multimedia framework
OpenCV	635.27	Computer Vision
OpenSSL	431.81	Network
GIFLib	421.06	Graphis
SQlite3	369.29	RDBMS
WebP	213.40	Codec

TABLE VIII: Top 10 most neglected vulnerable libraries in terms of the average time-to-fix

Finding 9: Future research should focus on these highly neglected libraries as experimental subjects for determining methods to ease the burden of updating these libraries; running regression tests to ensure these updates do not introduce new errors; and repairing those errors, possibly automatically, when they do arise.

D. Exploitability Case Study

To demonstrate the exploitability of unpatched vulnerabilities in third party apps, we carry out a targeted case study where we analyze individual applications and create a proof-of-concept (PoC) exploit. Our PoC highlights how these unpatched vulnerabilities can be exploited by third parties when interacting with the apps.

XRecoder allows users to capture screen videos, screen shots, and record video calls. Furthermore, XRecoder provides video editing functionalities, enabling users to trim videos and change their speed. This application uses FFmpeg, an open-source video encoding framework that provides video and audio editing, format transcoding, video scaling and post-production effects.

XRecoder embeds the FFmpeg library version 3.1.11, which is vulnerable to *CVE-2018-14394* (reported in July 2018). FFmpeg-3.1.11 contains a vulnerable function (*ff_mov_write_packet*) that may result in a division-by-zero error if provided with an empty input packet. Hence, an attacker can craft a WaveForm audio to cause denial of service.

To assess whether this vulnerable function is reachable in XRecoder, we used *Radare2* [34] to replace the first instruction in the vulnerable function with an interrupt instruction. We run the application after the latter modification which consequently resulted in an app crash, i.e., allowing us to trigger the vulnerability consistently.

`ff_mov_write_packet` is called by multiple functions across two different binaries (FFmpeg-3.1.11.so and the app-specific libisvideo.so) and two different platforms (Dalvik and Native). `av_buffersink_get_frame`, one of the ancestors of `ff_mov_write_packet`, is called by `nativeGenerateWaveFormData` from the Dalvik-side.

IV. Discussion:

Findings in *RQ2* (Section III-B) demonstrate that out of 7,253 binaries for which we inferred their versions, 3,674 were vulnerable libraries (50.65%) affecting 53 distinct apps between Sept. 2013 and May 2020. This constitutes about 26.5% of the top 200 apps on Google Play. More alarmingly, new releases of 14 distinct apps remain vulnerable even at the time at which we collected apps for this study with an average outdatedness of 859.17 ± 137.55 days. While we have informed app developers about the outdated libraries in their apps, one interesting piece of follow-up work based on this result is surveying Android app developers to determine the reason for this extremely slow rate of fixing vulnerable native libraries in their apps. Such a study can further assess what forms of support app developers would need to truly reduce this slow rate of updating vulnerable library versions to ones with security patches.

For RQ3 (Section III-C), we analyzed the speed at which developers updated their apps to patched libraries and found

that, on average, library developers release a security patch after 54.59 ± 8.12 days from a reported *CVE*. While app developers apply these patches on average after 528.71 ± 41.20 days from the date an update was made available (10 times slower). Recall that we only consider apps in these cases that actually ended up fixing vulnerable native libraries. The results for RQ2 and RQ3 corroborate the need to make app developers aware of the severe risks they are exposing their users to by utilizing vulnerable native libraries.

Overall, our results demonstrate the degree to which native libraries are neglected in terms of leaving them vulnerable. Unfortunately, our findings indicate that the degree of negligence of native libraries is severe, while popular apps on Google Play use native libraries extensively with 145 out of 200 top free apps (72.50%). Interesting future work for our study includes uncovering the root causes of such negligence and means of aiding developers to quickly update their native libraries. For example, platform providers (e.g., Google) could provide mechanisms to automatically update native libraries while also testing for regressions and possibly automatically repairing them. Such an idea is similar to how Debian's repositories centrally manage libraries and dependencies between applications and libraries. Whenever a library is updated, only the patched library is updated, the applications remain the same. The Android system would highly profit from a similar approach of central dependency and vulnerability management.

V. Threats to Validity

External validity. The primary external threat to validity involves the generalizability of the data set collection and the selection methodology. Recent changes in Google Play limited the length of the "top-apps" list to 200 items. Despite the restrictions imposed by Google Play (limiting our analysis to the top-200 apps), these apps (1) account for the bulk of downloads and the largest user base on Google Play and (2) are generalizable to popular apps, thus having the largest impact.

The results from *RQ1* show that *LibRARIAN* detects versions of native libraries with high accuracy (91.15%). The need to compare against binaries with a known number of versions and libraries (i.e., *Known Lib Versions* in Figure 1) limits *LibRARIAN*. Specifically, misidentification of a library or its version might occur when an unknown binary for which we are trying to identify a library and version does not exist in *Known Lib Versions*. In these cases, *LibRARIAN* identifies the unknown binary as being the library and version closest to it according to bin^2sim that exists in *Known Lib Versions*. One possible way of enhancing *LibRARIAN* in such cases is to leverage supervised machine learning, which may, at least, be able to identify if the library is most likely an unknown major, minor, or patch version of a known library.

Internal validity. One internal threat is the accuracy of timestamps in AndroZoo and its effect on the reported patch life cycle findings. To mitigate this threat, we collected AndroZoo timestamps over three months and correlated updates with Google Play. We verified that AndroZoo has a maximum lag of 9 days. This short delay is much smaller than the update frequency of vulnerable apps. Furthermore, we verified that using dates added to AndroZoo and version codes give us reliable timestamps for earlier time periods.

Construct validity. One threat to construct validity is the labeling of the libraries in our repository as vulnerable or not. To mitigate this threat, we relied on the vulnerabilities reported by the Common Vulnerabilities and Exposures database [7] which contains a list of publicly known security vulnerabilities along with a description of each vulnerability.

We conducted an exploitability case study of one vulnerable library in an app Section III-D. For the remaining set of discovered vulnerable libraries/apps, we verified that vulnerable native functions are exported and that the library is loaded from the app/Dalvik-side. Performing a complete analysis of exploitable/reachable native functions in Android is an interesting but orthogonal research problem. Building a cross-language control-flow/data-flow analysis to assess reachability of vulnerable native code from the Dalvik code of an Android app is an open research problem, worthy of a separate research paper: (1) recovering a binary CFG/DFG is currently unsound, based on heuristics, and runs into state explosion and (2) conducting an exploitability study of all vulnerable libraries/apps across our entire dataset is infeasible due to the large amount of apps/libraries.

Another threat to validity is the possibility of developers manually patching security vulnerabilities. To mitigate this threat to validity, we checked the versions identified by *LibRARIAN* and found that *LibRARIAN* correctly identifies an overwhelming majority of patch-level versions (61.21%). For the patch-level versions that *LibRARIAN* cannot distinguish as effectively, *LibRARIAN* makes manual identification much easier, by significantly reducing the search space for post analysis to only 3-4 candidate versions. Furthermore, based on the results of our dataset, we believe that app developers are unlikely to manually patch a library they do not maintain given that it already takes years for these developers to simply update a library version.

VI. Related Work

A series of work has demonstrated the importance of third-party libraries for managed code of Android apps (i.e., Dalvik code) and their security effects and implications [9], [4]. Derr et al. [9] investigated the outdatedness of libraries in Android apps by conducting a survey with more than 200 app developers. They reported that a substantial number of apps use outdated libraries and that almost 98% of $17K$ actively used library versions have known security vulnerabilities. Backes et al. [4] report, for managed code-level libraries, that app developers are slow to update to new library versions—discovering that two long-known security vulnerabilities remained present in top apps during the time of their study. None of these studies examined native third-party libraries in Android apps nor did they look at the security impact of vulnerable libraries or whether these vulnerabilities are on the attack surface. *LibRARIAN* now explores the attack surface of

native libraries, closing this important gap and calling platform providers to action.

A wide variety of approaches have emerged that identify third-party libraries with a focus on managed code. These approaches employ different mechanisms to detect third-party libraries within code including white-listing package names [17], [5]; supervised machine learning [32], [27]; and code clustering [41], [28], [23]. LibScout [4] proposed a different technique to detect libraries using normalized classes as a feature that provides obfuscation resiliency.

Some techniques identify vulnerabilities in native libraries by computing a similarity score between binaries with known vulnerabilities and target binaries of interest [15][12]. VulSeeker [15] matches binaries with known vulnerabilities using control-flow graphs and machine learning. Similarly, discovRE [12] and BinXray [42] matches binaries at the function level. Other techniques employ a hybrid technique such as *BinSim*[30], *Mobilefinder*[24], *BinMatch*[21], and *DroidNative* [1]. These approaches identify semantic similarities/differences between functions in binaries based on execution traces for the purpose of analyzing/identifying malware. Unlike these tools, *LibRARIAN* focuses on benign libraries with the goal of identifying their versions with high scalability.

Binary Analysis Tool (BAT) [19] and *OSSPolice* [10] measure similarity between strings extracted from binaries and features found directly in source repositories. Unlike *LibRARIAN*, these approaches compare source code with binaries, which introduces the issue of internal clones (i.e., third-party library source code that is reused in the source code of another library). BAT and *OSSPolice* rely on simple syntactic features (e.g., string literals and exported functions). *OSSPolice* cannot detect internal code clones, while *LibRARIAN* can, giving it superior ability to identify versions of native libraries. Furthermore, BAT does not detect versions of binaries and was shown to have inferior accuracy for computing binary similarity compared to *OSSPolice*. Unlike these tools, *LibRARIAN* extracts additional features—such as imported functions, exported and imported global variables, and dependencies that uniquely identify different versions of binaries. As shown in Section III-A1, these additional features were a major factor in the superior accuracy of *LibRARIAN* compared to *OSSPolice*.

Other related empirical research studies the prevalence of vulnerable dependencies in open source projects [6], vulnerabilities in WebAssembly binaries [22], or investigates the updatability of ad libraries in Android Apps [31]. Other work such as [18], [43] present third party library recommendation tools for mobile apps.

Despite the existence of much previous work on survivability of vulnerabilities in Android apps/libraries, such work has not conducted a large-scale longitudinal study of native third-party libraries as we did in this paper. Moreover, the survivability of vulnerabilities in non-native libraries are significantly shorter compared to those reported in our results. While survivability of vulnerabilities in native Android apps took, on average, 528.71 ± 40.20 days in our study, prior work [3], [29] shows that survival times of vulnerabilities in Python and Javascript are 100 days and 365 days, respectively. 50% of vulnerabilities in npm packages were fixed within a month, 75% were fixed within 6 months only [8].

None of this aforementioned related work has examined the prevalence of security vulnerabilities in Android's native libraries or the time-to-fix for vulnerable versions of such libraries. As a result, our work covers a critical attack vector that has been ignored in existing research.

VII. Conclusion

Third-party libraries have become ubiquitous among popular apps in the official Android market, Google Play, with 145 out of the 200 top free apps on Google Play (72.50%) containing native libraries. These libraries are particularly beneficial for handling CPU-intensive tasks and for reusing existing code in general. Unfortunately, the pervasiveness of native third-party libraries in Android apps expose end users to a large set of unpatched security vulnerabilities.

To determine the extent to which these native libraries remain vulnerable in Android apps, we study the prevalence of native libraries in the top 200 apps on Google Play across 7,253 versions of those apps. From these versions, we extracted 66,684 native libraries. To identify versions of libraries, we constructed an approach called *LibRARIAN* that leverages a novel similarity metric, bin^2sim, that is capable of identifying versions of native libraries with a high accuracy—a 91.15% correct identification rate.

For vulnerabilities, we found 53 apps with vulnerable versions with known CVEs between Sept. 2013 and May 2020, with 14 of those apps still remaining vulnerable until the end point of our study. We find that app developers took, on average, 528.71 ± 40.20 days to apply security patches, while library developers release a security patch after 54.59 ± 8.12 days—a 10 times slower rate of update.

VIII. Data Availability

Our dataset, analysis platform, and results are available online [25] for reusability and reproducibility purposes.

IX. Acknowledgement

This work was supported in part by award CNS-1823262 and CNS-1801601 from the National Science Foundation and grant number 850868 from the ERC Horizon 2020 program.

REFERENCES

[1] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. DroidNative: Automating and optimizing detection of android native code malware variants. *Computers & Security*, 65:230 – 246, 2017.

[2] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: collecting millions of android apps for the research community. In *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*, pages 468–471. ACM Press, 2016.

[3] Gábor Antal, Márton Keleti, and Péter Hegedundefineds. Exploring the security awareness of the python and javascript open source communities. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 16–20, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, pages 356–367. ACM Press, 2016.

[5] Theodore Book, Adam Pridgen, and Dan S. Wallach. Longitudinal analysis of android ad library permissions. *arXiv:1303.0857 [cs]*, 2013.

[6] Brandon Carlson, Kevin Leach, Darko Marinov, Meiyappan Nagappan, and Atul Prakash. Open source vulnerability notification. In Francis Bordeleau, Alberto Sillitti, Paulo Meirelles, and Valentina Lenarduzzi, editors, *Open Source Systems - 15th IFIP WG 2.13 International Conference, OSS 2019, Proceedings*, IFIP Advances in Information and Communication Technology, pages 12–23. Springer New York LLC, January 2019. 15th International Conference on Open Source Systems, OSS 2019 ; Conference date: 26-05-2019 Through 27-05-2019.

[7] Cve. https://cve.mitre.org/.

[8] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 181–191, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*, pages 2187–2200. ACM Press, 2017.

[10] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke 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 - CCS '17*, pages 2169–2185. ACM Press, 2017.

[11] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *in Proceedings of the 20th USENIX Security Symposium*, page 16, 2011.

[12] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovRE: Efficient cross-architecture identification of bugs in binary code. In *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, 2016.

[13] F-droid. https://f-droid.org.

[14] Ffmpeg. https://ffmpeg.org/.

[15] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*, pages 896–899. ACM Press, 2018.

[16] gnu.org. https://developer.android.com/ndk.

[17] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks - WISEC '12*, page 101. ACM Press, 2012.

[18] Qiang He, Bo Li, Feifei Chen, John Grundy, Xin Xia, and Yun Yang. Diversified Third-party Library Prediction for Mobile App Development. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[19] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *Proceeding of the 8th working conference on Mining software repositories - MSR '11*, page 63. ACM Press, 2011.

[20] Hewlett packard enterprise cyber risk report 2016. https://www.thehaguesecuritydelta.com/media/com_hsd/report/57/document/4aa6-3786enw.pdf.

[21] Yikun Hu, Yuanyuan Zhang, Juanru Li, Hui Wang, Bodong Li, and Dawu Gu. BinMatch: A semantics-based hybrid approach on binary code clone analysis. *arXiv:1808.06216 [cs]*, 2018-08-19.

[22] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of webassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234. USENIX Association, August 2020.

[23] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo. LibD: Scalable and precise third-party library detection in android markets. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 335–346, 2017.

[24] Yibin Liao, Ruoyan Cai, Guodong Zhu, Yue Yin, and Kang Li. MobileFindr: Function similarity identification for reversing mobile binaries. In Javier Lopez, Jianying Zhou, and Miguel Soriano, editors, *Computer Security*, volume 11098, pages 66–83. Springer International Publishing, 2018.

[25] Librarian. https://github.com/salmanee/Librarian.

[26] libvpx. https://github.com/webmproject/libvpx/.

[27] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh 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 - MobiSys '15*, pages 89–103. ACM Press, 2015.

[28] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. LibRadar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*, pages 653–656. ACM Press, 2016.

[29] Alejandro Mazuera-Rozo, Jairo Bautista-Mora, Mario Linares-Vásquez, Sandra Rueda, and Gabriele Bavota. The android os stack and its vulnerabilities: an empirical study. *Empirical Software Engineering*, 24(4):2056–2101, 08 2019. Copyright - Empirical Software Engineering is a copyright of Springer, (2019). All Rights Reserved; Last updated - 2019-07-25.

[30] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *USENIX Security Symposium*, 2017.

[31] Israel J. Mojica Ruiz, Meiyappan Nagappan, Bram Adams, Thorsten Berger, Steffen Dienst, and Ahmed E. Hassan. Analyzing Ad Library Updates in Android Apps. *IEEE Software*, 33(2):74–80, March 2016. Conference Name: IEEE Software.

[32] 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)*, pages 1–6, 2017.

[33] Opus. https://opus-codec.org/.

[34] Radare2. https://rada.re/n.

[35] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. FLEXDROID: Enforcing in-app privilege separation in android. In *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, 2016.

[36] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2017.

[37] Sonatype - 2019 state of the software supply chain. https://www.sonatype.com/2019ssc.

[38] Speex. https://www.speex.org/.

[39] Mengtao Sun and Gang Tan. NativeGuard: protecting android applications from third-party native libraries. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks - WiSec '14*, pages 165–176. ACM Press, 2014.

[40] Vorbis. https://xiph.org/vorbis/.

[41] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun 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 - ISSTA 2015*, pages 71–82. ACM Press, 2015.

[42] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 376–387. ACM, July 2020.

[43] Huan Yu, Xin Xia, Xiaoqiong Zhao, and Weiwei Qiu. Combining Collaborative Filtering and Topic Modeling for More Accurate Android Mobile App Library Recommendation. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware - Internetware'17*, pages 1–6, Shanghai, China, 2017. ACM Press.

[44] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Network and Distributed System Security Symposium*, page 13, 2012.