Too Quiet in the Library: A Study of Native Third-Party Libraries in Android

Sumaya Almane
Department of Informatics
Institute for Software Research
University of California, Irvine
salmanee@uci.edu

Mathias Payer
School of Computer and Communication Sciences
École polytechnique fédérale de Lausanne
mathias.payer@nebelwelt.net

Joshua Garcia
Department of Informatics
Institute for Software Research
University of California, Irvine
joshug4@uci.edu

Abstract—Android applications (“apps”) make avid use of third-party native libraries to increase performance and to reuse already implemented functionality. Native code can be directly executed from apps through the Java Native Interface or the Android Native Development Kit. Android developers drop precompiled native libraries into their projects, enabling their use. Unfortunately, developers are often not aware that these libraries (or their dependencies) must be updated. This results in the continuous use of outdated native libraries with unpatched security vulnerabilities years after patches are available.

To assess the severity of the use of outdated and vulnerable libraries in the Android ecosystem, we study the prevalence of native libraries in the top applications of the Google Play market over time, correlating the time when native libraries are updated with the availability of security patches. A core difficulty we have to solve for this research is the identification of libraries and versions. Developers often rename or modify libraries but we require precise information about each binary. Our binary similarity metric $bin^2sim$ uses diverse features extracted from the libraries to identify and map the required information. Leveraging this $bin^2sim$, we create an approach called LibRARIAN (LibRary veRsion IdentificAtioN) that can accurately identify native libraries and their versions as found in Android apps with a 92.53% true-positive rate, no false positives, and a 7.46% false-negative rate.

In our study using LibRARIAN, we find that many libraries are outdated and that security patches are applied with long delays, if at all. We discovered that native libraries in apps are updated, on average, 3 times slower than the release rate of new versions of those libraries. For vulnerabilities, we found 80 apps with 1,781 vulnerable versions with known CVEs between Sept. 2013 and April 2019, with 61 of those apps still remaining vulnerable until the end point of our study. We find that app developers took, on average, 507.21±70.97 days to apply security patches, while library developers release a security patch after 19.04 ± 14.35 days—a 27 times slower rate of update.

I. INTRODUCTION

Third-party libraries are an integral part of the development of mobile apps. Android developers opt for third-party libraries due to their convenience and re-usability, since utilizing them saves time and effort and allows developers to avoid re-implementing functionality. Furthermore, native libraries have become more prevalent in recent Android applications (“apps”), especially social networking and gaming apps. These two categories of apps—which ranked among the top categories on Google Play—require special tasks such as 3D rendering, and encoding and decoding of audio and video.

These tasks tend to be hardware-intensive and are, thus, often handled by native libraries to improve runtime performance.

Despite the convenience and benefits that mobile developers obtain from third-party libraries, they can expose end-users to a wide range of security attacks. For example, in August 2019, Kaspersky found that an app called CamScanner, with more than 100 million downloads on Google Play, shipped with a third-party ad library containing a malicious module that signs users up for paid subscriptions [6]. Some other malicious ad libraries have affected about 440 million Android users of Google Play [11].

Prior work [24], [26] has shown that the ubiquity of third-party libraries in Android apps increases the attack surface since host apps expose vulnerabilities propagated from these libraries [3], [7]. Another series of previous work has studied the outdatedness and updateability of third-party Java libraries in Android apps [13], [10], 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 do not keep them updated. The reasons for this are manifold including concerns over regressions arising from such updates, prioritizing new functionality over security, deadline pressures, and other forms of negligence (such as a lack of tracking library dependencies and their security patches) that 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 very difficult to exploit with managed code of Android apps, i.e., Dalvik code. Third, and contrary to previous studies [15], [28], native libraries are currently used pervasively in top mobile apps. To illustrate this point, Table I shows the pervasiveness of native libraries in the top 600 free apps collected from Google Play between Sept. 2013 and April 2019. We obtained the version histories of these apps from AndroZoo [8] totalling 12,646 versions of those 600 top free apps. On average, there are 21 versions per app with as few as 1 version per app and as many as 136 for com.twitter.android. From these apps, we identified 89,525 native libraries in total with an average of 8 libraries per app—with as few as 1 native library per app and as many as 141.

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 (LibRARy veRsion IdentificAtioN), that takes an unknown native binary and identifies (i) the library it implements and (ii) the library’s version. We have demonstrated this approach to be scalable to nearly 90,000 versions of apps, allowing us to identify vulnerable and updated versions of such apps.

- We introduce a new similarity-scoring mechanism for comparing native binaries called $\text{bin}^2\text{sim}$, which utilizes 8 feature vectors that enable LibRARIAN to distinguish between different libraries and their versions. These features represent the code elements of a library that would be expected to change based on a versioning scheme that distinguishes between major, minor, and patch versions of a native library.

- We build a repository of Android apps and their native libraries that contains the 600 most popular free apps from Google Play totalling 12,646 versions gathered between the dates of Sept. 2013 and April 29, 2019. This repository further contains 89,525 native libraries used by these 12,646 versions.

- By leveraging LibRARIAN, $\text{bin}^2\text{sim}$, and our repository, we conduct a study of the accuracy of LibRARIAN, finding that on 3,907 binaries with 81 distinct versions, LibRARIAN obtains a 92.53% true-positive rate, no false positives, and a 7.46% false-negative rate.

- We compare LibRARIAN with a state-of-the-art native-library version-identification approach called OSSPolice [14]. We demonstrate on OSSPolice’s ground-truth dataset an over 10% accuracy improvement without having to rely on source code—OSSPolice requires source code to identify versions, limiting its applicability for Android apps.

- We utilize LibRARIAN to study the outdatedness of native libraries. From 10,018 library instances—which serve as accurate representations of library versions using $\text{bin}^2\text{sim}$—we find that third-party libraries remain unchanged, on average, for 252.12 ± 29.40 days with an app’s native library being outdated as long as 1,147 days, while new releases of these libraries are made available every 83.05 ± 18.49 days. As a result, many top free apps on Google Play remain vulnerable for very long periods of time.

- Using LibRARIAN, we examine 80 apps with 1,781 vulnerable versions with known CVEs between Sept. 2013 and April 2019. 61 of these apps remain vulnerable up until the end of our data collection—indicating that many of these apps are likely to remain vulnerable to this day. We further find that these apps have a long period of outdatedness, on average, of 628 ± 73.4 days.

- By utilizing LibRARIAN, we analyzed 19 apps with 317 versions, focusing on vulnerable versions of FFmpeg, GIFLIB, OpenSSL, WebP, SQLite, and OpenCV between Sept. 2013 and April 2019. We find that app developers took, on average, 507.21 ± 70.97 days to apply security patches, while library developers release a security patch after 19.04 ± 14.35 days—a 27 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.

## II. LibRARIAN

Figure 2 shows the overall workflow of LibRARIAN which enables identification of native libraries and their versions. While our approach is general, we analyze native libraries used in Android apps. Given a set of native binaries with unknown library names and versions extracted from Android apps (Unknown Lib Versions in Figure 2) and a set of native binaries with known libraries and versions (Known Lib Versions in Figure 2), LibRARIAN identifies libraries and outputs a final set of versions detected for third-party native libraries of Android apps by matching the unknown versions with known versions.

LibRARIAN begins by performing an initial Library Identification as depicted in Figure 2, which utilizes the naming convention for shared libraries described in the Linux Documentation Project [5]. This convention states that naming shared objects must start with the prefix “lib” followed by the name of the library and the extension “.so” and ending with a version number reflecting the incremental changes in the library (e.g., libreadline.so.3.0 for version 3.0 of the readline library). Library Identification parses the name of the file by eliminating the “lib” prefix, “.so” extension, and version numbers to obtain an initial library name (e.g., “readline”). Although developers do not always utilize such a naming convention, they often include some part of the correct name of the library (e.g., readline, opencv, etc.), which our evaluation (see Section V) corroborates, allowing LibRARIAN to potentially perform far fewer comparisons.

One of the key challenges of identifying many versions of Android-app native binaries is reducing the number of duplicate native binaries (i.e., Binary Duplicate Elimination in Figure 2). In particular, such a step makes manually verifying the accuracy of LibRARIAN’s version identification substantially more feasible. As an example, from a total of 89,525 native binaries extracted from multiple versions of the top 600 apps on Google Play, LibRARIAN reduced this number to 10,018 unique binaries—a factor of 8.93x.

To eliminate duplicate native binaries, LibRARIAN first clusters any native binaries that share the same sha256 code into the same cluster. This Hash Code Clustering, as shown in Figure 2, produces a set of unique library instances, i.e.,

---

| Table I: Prevalence of native libraries in the top 600 apps on Google Play |
|-----------------------------|-----------------|-----------------|
| Num. of apps containing native libs | 540/600 |
| AVG. versions per app | 21 versions |
| MAX. versions per app | 136 |
| MIN. versions per app | 1 |
| AVG. native libs per app (including all app versions) | 8 |
| MAX. native libs per app (including all app versions) | 141 |
| MIN. native libs per app (including all app versions) | 1 |

---

1The majority of our analyses are up-to-date and contiguous but, e.g., the top 600 apps continuously change. We used April 29, 2019 as a cutoff, as building a manual ground-truth data set (collecting library binaries with known versions from different sources) to validate our results takes a considerable amount of time.
Library Instances. LibRARIAN then performs a second type of clustering, which we refer to as Bin²Sim Clustering, that creates clusters by analyzing binaries in significantly more detail to further reduce duplicates. This clustering results in another intermediate set of library instances, i.e., Library Instances₁₂, in Figure 2. Bin²Sim Clustering leverages our novel technique for computing the similarity between native binaries called bin²sim. Note that Hash Code Clustering is an optimization step which reduces the runtime needed to perform Binary Duplicate Elimination. Bin²Sim Clustering can eliminate all duplicates that Hash Code Clustering eliminates.

In the final step, Version Identification determines the versions of binaries in Library Instances₁₂ by leveraging bin²sim to match those instances with binaries in Known Lib Versions.

III. bin²sim

To properly identify library versions, LibRARIAN relies on bin²sim to perform Bin²Sim Clustering and Version Identification as described in the previous section. bin²sim (i) generates feature vectors from native binaries and (ii) computes a similarity score between source and target feature vectors. Prior to elaborating on these two major elements of bin²sim, we first discuss the manner in which native binaries work in Android apps.

A. Feature Vector Extraction

All shared libraries included in Android apps are compiled into Executable and Linkable Format (ELF) binaries. Like any other object files, ELF binaries contain a symbol table of externally visible identifiers such as function names, global symbols, and local symbols. Many GNU binutils tools (e.g., objdump, nm, and strings) and binary analysis frameworks (e.g., angr [25]) refer to the symbol table to reverse engineer an executable.

To distinguish between different libraries and their versions, we need to identify features, which allow LibRARIAN to differentiate between instances of a particular library and its various versions. To that end, we define a set of 8 features that LibRARIAN uses to identify versions and libraries. Table II shows the list of all features used by bin²sim. The primary collected features include the following: exported and imported functions, exported and imported globals, and library dependencies. These 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 available across platforms regardless of the underlying architecture or compilation environments.

In general, bin²sim’s matching algorithm takes these five features into account when computing the similarity score between app binaries and source binaries. For libraries where the functions are called through a dispatcher function (e.g., the RenderScript and Unity libraries export a single function that, based on runtime parameters, dynamically invokes the desired functionality), these features fail to provide any sufficient information about the underlying components in a library. In such cases, additional features are considered as a second factor.

These features include strings that reside in the .comment section of the ELF symbol table, strings associated with specific keywords like “version” or Java package names, and certain debug strings and commands. The AllStrings feature is used for debug purposes only, it is not considered when computing the similarity score. Any symbols that are volatile across different architectures and build environments like compiler internals, relocatable entries, and debug symbols are not extracted.

B. Similarity Computation

bin²sim relies on the Jaccard coefficient to determine the similarity between feature vectors. The Jaccard coefficient allows bin²sim to account for addition or removal of features among different versions of the same library. Given two binaries b₁ and b₂ with respective feature vectors FV₁ and FV₂, the Jaccard coefficient is the size of the intersection of FV₁ and FV₂ (i.e., the number of common features) over the size of the union of FV₁ and FV₂ (i.e., the number of unique features):

\[ \text{JaccardSim}(FV₁, FV₂) = \frac{|FV₁ \cap FV₂|}{|FV₁ \cup FV₂|} \quad \epsilon [0, 1] \] (1)

The similarity score is a floating-point value between 0 and 1, with a score of 1 indicating identical features, and a score of 0 indicating no shared features between the two libraries. We classify a library as partially matching if the score drops below 1, this implies that only a subset of the features included in the feature vectors match.

Bin²Sim Clustering (as shown in Figure 2) groups binaries into the same cluster if their Jaccard coefficient matches exactly (i.e., scores a 1). In such cases, the matches must be exact since the goal is to remove duplicates for that step of LibRARIAN. On the other hand, Version Identification counts an unknown library instance from Library Instances₁₂ as matching a known library version if their Jaccard coefficient is above 0.85. This threshold was determined experimentally and works effectively as our evaluation will demonstrate (see Section V).

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 (e.g., 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 arm64-v7a and x86 export more features compared to arm64-v8a and x86_64.

IV. APPLICATION AND LIBRARY REPOSITORY

To study the security implications of the usage of third-party native libraries, we apply our approach to libraries
Fig. 1: LibRARIAN identifies versions of native binaries from Android apps by using our \(bin^2sim\) similarity-scoring technique to compare known and unknown versions of native binaries.

| Feature Name | Feature Definition |
|--------------|--------------------|
| Global Variables | Variables defined in a library that are either linked to other libraries or imported by them |
| Imported Globals | Variables accessible to a library and originating from other libraries |
| Exported Functions | Functions defined in a library that are either linked to other libraries or imported by them |
| Imported Functions | Functions accessible to a library and originating from other libraries |
| Dependencies | The library dependencies that are automatically loaded by the ELF object |
| AllStrings | A string dump of the `.rodata` section of an ELF object which contains read-only initialized data |
| Debug Strings | Strings obtained from the `.comment` section of an ELF object which represents the version used to compile the binary and its execution environments |
| String Identifiers | A subset of AllStrings which includes any string that parses as an identifier (e.g., mangled C++ identifiers). This feature also contains keywords such as "version". |

TABLE II: List of features \(bin^2sim\) extracts from native binaries of Android apps

Fig. 2: \(bin^2sim\) takes two binary files (e.g., one extracted from an Android .apk file and another compiled from source code) and obtains a similarity score using either the Jaccard coefficient or the overlap coefficient. An app update on average every 17 days. We determined that the developers of the apps in our dataset may have been added to AndroZoo. For these release dates, which an APK was added to AndroZoo to estimate updates of longitudinal study, we use the timestamp denoting the time at which an APK was added to AndroZoo to estimate updates of all apps. The average duration between an app’s earliest release date in our dataset and its latest release date is 833 ± 30.43 days—with a maximum of 2,146 days (≈ 5.8 years) for the app \(com.heartbeat.expressionweather\). We optimized the latter approach by utilizing the length of features. This optimization reduced the required time to 704 hours while the total time to further cluster the resulting libraries extracted from Android apps for which we will apply LibRARIAN to identify libraries and detect versions. The first task is to reduce the number of native libraries by removing any duplicate files. Recall that we found \(≈ 90\%\) native libraries in the top 600 apps of Google Play. After applying Hash Code Clustering from Figure 2 on extracted libraries, we reduce their number from 89,525 to 18,300 Library Instances. The next step to reduce duplicate binaries (BinSim Clustering in Figure 2) decreases the number of Library Instances to 10,018.

We run LibRARIAN on a machine with 2 AMD EPYC 7551 32-Core CPUs and 512GB of RAM running Ubuntu 18.04. The total execution time for Hash Code Clustering is 2.5 hours while the total time to further cluster the resulting Library Instances using BinSim Clustering is 1.5 hours. We optimized the latter approach by utilizing the length of feature vectors, i.e., we avoid computing the \(bin^2sim\) between two feature vectors unless both of them have the same number of features. This optimization reduced the required time to produce Library Instances from 8 months to 1.5 hours.

The average number of features in the extracted feature vectors (excluding AllStrings and Debug Strings) is 6,014.81 ± 200 features. Some outliers such IL2CPP (Unity’s scripting
LibRARIAN library and UE4 (Unreal Engine 4) library include 389,345 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. For such complex binaries, the average time to generate the feature vector is 4 min and 38 sec.

V. Evaluation

To conduct our study of native libraries in Android apps, we answer the following four research questions:

RQ1: How accurate is LibRARIAN at identifying versions of native libraries? How does LibRARIAN compare against state-of-the-art native-library version identification?

RQ2: How outdated are native libraries of Android apps?

RQ3: How prevalent are vulnerabilities in native libraries of Android apps?

RQ4: In case a vulnerability was reported for a third-party library, how quickly did developers apply patches?

A. RQ1: Accuracy

1) Independent Accuracy: In order to evaluate the accuracy of LibRARIAN, we select the subset of native libraries from our library repository (as described in Section IV) that contain the exact version number of a library in the string literals. In total, there are 3,907 binaries (268 Library Instances*) in our repository. After eliminating duplicates, we obtained 81 distinct versions of 18 libraries that serve as ground truth for this analysis (Table III). We then apply LibRARIAN to these binaries to assess its accuracy based on our bin²sim feature comparison (Section III-B).

| Library Name   | Library Versions            |
|----------------|------------------------------|
| Crashlytics    | 0.5.0, 1.0.0, 1.1.0, 1.1.1-1.1.5, 2.0.0-2.0.5 |
| OpenCV         | 2.4.1, 2.4.13, 3.1.0         |
| OpenCV_core    | 2.4.11, 2.4.13              |
| OpenCV_imgproc | 2.4.11, 2.4.13              |
| LibVPX         | 1.3.0-1.7.0                 |
| Firebase       | 5.1.0-5.6.0                 |
| AVCodec        | 54.92.100, 55.18.102, 55.39.101, 55.52.102, 56.1.100, 57.64.101 |
| AVFilter       | 3.90.100, 4.2.100, 5.1.100, 6.65.100 |
| AVFormat       | 55.19.104, 55.33.100, 56.4.101, 57.56.101 |
| AVUtil         | 52.18.100, 52.48.101,52.66.100,54.20.100, 55.34.101 |
| swscale        | 2.0.101, 0.17.104           |
| swresample     | 2.0.101, 0.17.104           |
| Vorbis         | 1.3.2, 1.3.3                |
| XML2           | 2.7.7                       |
| OpenSSL        | 1.12.854, 1.15.1, 1.18.2    |
| SQLite3        | 3.11.0, 3.13.0, 3.14.1, 3.20.1, 3.24.0-3.26.0, 3.6.22, 3.7.17, 3.8.1-3.8.10 |
| Mono           | 4.6, 4.6.3, 5.5             |

TABLE III: The list of 18 libraries (82 different versions) used to evaluate the accuracy results of LibRARIAN

LibRARIAN correctly identifies 95.66% of the unique versions (77 out of 81). Out of these 77 versions, 51 (66.23%) have unique feature vectors—resulting in perfect matches of unknown versions to known versions. The remaining 26 versions share similar feature vectors with a maximum of 1-2 other versions. This usually occurs between consecutive versions—usually 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.

We inspected the binaries reported with false positives and found that our approach failed to detect the correct versions of 3 binaries due to the fact that each of their target binaries is missing from our ground truth (as shown in Figure 2). In such cases, the version reported is the version of an existing binary in the ground truth that is closest to the target binary.

One false negative (1%) is reported in our results—where the target version exists in our repository yet the reported score is low (< 5%). When we inspected the binary, we found that it was modified by the app developers resulting in a much smaller feature vector. Recall from Section III-B that bin²sim penalizes target and source binaries if they do not contain the exact same features.

2) Comparative Analysis: OSSPolice compares binaries against source code to identify versions of the binaries, requiring source code of the target libraries to build its index. We repeatedly contacted the authors of OSSPolice but were unable to obtain their non-public data index or sufficient information to reproduce their setup. We performed a comparative analysis between LibRARIAN and OSSPolice based on OSSPolice’s published numbers [14]. The ground-truth data set in the OSSPolice evaluation contains a total of 475 binaries (out of which 67 are unique) extracted from 104 applications collected by F-Droid [2]. Table IV highlights the evaluation of OSSPolice and LibRARIAN on the same data set.

| Approach       | # Versions | Uniq. Bins | True +ve | False +ve | False -ve |
|----------------|------------|------------|----------|-----------|-----------|
| LibRARIAN       | 172        | 67         | 62 (92.53%) | 5 (7.46%) |
| OSSPolice       | 172        | 67         | 55 (82%)  | N/A        | N/A        |

TABLE IV: Comparing accuracy of LibRARIAN with OSSPolice on the OSSPolice data set.

LibRARIAN correctly identified 62/67 (92.53%) unique binaries, improving precision by over 10% compared to the accuracy reported by OSSPolice (82%). One binary was not detected due to the incompleteness of our source repository, while the remaining 4 were not identified because the library functions are dispatched from a single function, hence, our extracted features fail to provide sufficient information about the underlying components in the library.

OSSPolice’s reported false positives and negatives are due to the fact that it relies on simple syntactical features such as string literals and exported functions. Our feature vectors contain additional features such as imported functions, exported and imported global variables, and dependencies that uniquely identify different versions of binaries.

Furthermore, OSSPolice fails to detect internal clones (i.e., third-party library source code that is reused in the source code of another library) as it heavily relies on the hierarchy of OSS folders. bin²sim’s strict similarity metric (i.e., the Jaccard coefficient) is resilient against this over-fitting to specific names.

B. RQ2: Outdatedness of Libraries in Top Apps

To study the outdatedness of native libraries in Android apps, we need to infer the versions of libraries in our repository (see Section IV) in order to analyze when library version lvₓ in app a is updated to library version lvᵧ, whether lvₓ is
Experiment Setup. Recall from Section II that LibRARIAN requires a set of library binaries with known versions (Known Lib Versions in Figure 1) to compare with binaries extracted from Android apps (Unknown Lib Versions in Figure 1) for Version Identification in Figure 1. However, manually identifying library versions for all binaries in our repository is practically infeasible because of the large size of our repository, which contains 89,525 binaries in total. The manual process of searching for and matching library versions for each binary—including building the library version from source or identifying pre-built binaries from existing repositories or the Web—would be intractable.

Nevertheless, we are interested in studying the outdatedness for as much of our repository as possible (i.e., all 89,525 binaries). To that end, we apply this study on Library Instances$_{02s}$, instead of exact library versions. Recall from Section IV that bin$^2$sim eliminates duplicate binaries resulting in 10,018 Library Instances$_{02s}$. Each library instance in Library Instances$_{02s}$ is a cluster of binaries that share the exact same feature vector as used in LibRARIAN from Section V-A.

We extracted 89,525 binaries from 540 unique Android apps with $\approx$ 10k app versions. Next, we apply Hash Code Clustering to eliminate duplicate binaries resulting in 18,300 Library Instances$_{hc}$. We then perform Bin$^2$Sim Clustering which further reduces the number of binaries to 10,018 Library Instances$_{02s}$. Each library instance in Library Instances$_{02s}$ is associated with the name and version of the app from which we extracted the binary. Using that information and release dates of apps, we calculated the average outdatedness of our entire library repository.

Using Library Instances$_{02s}$ instead of library versions allows us to study the outdatedness of thousands of binaries instead of only hundreds. However, to allow us to study outdatedness at the scale of all our repository’s binaries requires a trade-off in accuracy: It is possible that different library versions end up having the same feature vector. As a result, the feature vectors of two binaries can be equal if changes made between consecutive versions (e.g., version 3.0.0 and 3.0.1) of the app, the native library is updated—which is a conservative assumption. We found this result to affect no more than 14% of our results. For future work, we will look into a more fine-grained form of version identification for these binaries, allowing us to pinpoint and identify bug fixes in the binaries.

Table V illustrates the extent of outdated native libraries found in our app repository. We obtained this list by sorting Library Instances$_{02s}$ based on the number of years a Library Instance$_{02s}$ remains unchanged, starting with the most outdated Library Instance$_{02s}$ along with the name of the app from which we extracted this library. Table V includes the top 30 out of 10,018 Library Instances$_{02s}$ in terms of the number of years a Library Instance$_{02s}$ remained outdated. We compute the outdatedness of a Library Instance$_{02s}$ as the difference between (i) the time a specific Library Instance$_{02s}$ was first seen in an Android app and (ii) the last time it was last seen before it was replaced by another Library Instance$_{02s}$ or completely removed. Other factors in Table V include the number of library updates that were available during that period of library outdatedness and how many of those updates were security patches. We obtain library statistics about releases from the official library providers’ websites. For the rest of this section, we refer to library releases that add new features or fix bugs as regular updates, while we refer to library releases with security patches as security updates.

To demonstrate the prevalence of outdated libraries in our repository, we highlight a few examples from Table V. We start with Instagram—the app with the largest number of app versions in our list. Instagram includes a Library Instance$_{02s}$ of libvpx that was first seen in the oldest version of Instagram which dates back to January 2014 and was last seen in a recent version dating back to November 2018. As a result, libvpx was unchanged for 4.84 years across 96 different versions of Instagram. During that period, 5 regular updates of libvpx were available—none of which were used by Instagram.

SQLite released 33 library updates—which had the most number of library updates among the remaining libraries—between March 2016 and December 2018. 3 of those updates were security related. During that same window, the navigation app Waze used only one instance of libSQLite across 26 of its versions. The same applies to libGPG—which remained stale in two applications Sniper 3D and Lords Mobile for almost 3 years despite the fact that 28 new releases were made available during that period including 2 security updates.

The average outdatedness of OpenCV, which was found in 4 popular apps (Lyft, Ubercab, Groupon and PayPal), is $\approx$ 3 years. The developers of OpenCV release, on average, 6
new versions per year. This release rate indicates that these 4 popular apps that use OpenCV missed 18 opportunities to update that library to newer versions during the span of 3 years. 3 of those 18 newer versions contained patches for CVEs.

Besides assessing outdatedness of a particular library in an app, we also analyze the slow update of libraries of an Android app across all of its libraries. Table VI lists the top 10 apps in our dataset with the most outdated native libraries, measured based on the average number of days during which libraries of an app remained outdated. These apps have at least 5 million installs—with one app, Netflix, having over 500 million installs. The average outdatedness for these apps range from 529.90 days for Waze to as many as 1,147 days (almost 3 years) in the case of Progressive. The second-most neglected app is Flow Free, which has an average outdatedness of 1,029 days (2.8 years) across all of its libraries. Netflix which has the largest number of installs (500M+) in Table VI included libraries that remained outdated for 921 days (2.5 years).

| App Name               | Genre         | Num. Installs | Avg. Outdatedness (Days) |
|------------------------|---------------|---------------|--------------------------|
| Progressive            | Finance       | 5M+           | 1147.00                  |
| Flow Free              | Puzzle        | 100M+         | 1029.64                  |
| YouVersion Bible      | Books         | 100M+         | 999.00                   |
| HBO GO                 | Entertainment | 500M+         | 921.33                   |
| Lyft                   | Maps          | 10M+          | 873.00                   |
| Chase Mobile           | Finance       | 10M+          | 773.00                   |
| Yelp                   | Travel & Local| 10M+          | 602.83                   |
| Firefox                | Communication | 100M+         | 535.00                   |
| Waze                   | Maps          | 100M+         | 529.20                   |

**TABLE VI:** The top 10 apps with the most outdated native libraries

| Library Name | Avg. Outdatedness Across All Apps (days) | Num. Releases Per Year | Rate of Library Release (days) | No. of Apps Using Lib |
|--------------|------------------------------------------|------------------------|-------------------------------|------------------------|
| libunwind    | 1167                                     | 25                     | 240                           | 2                     |
| libgd         | 1114                                     | 6                      | 180                           | 2                     |
| libopenal    | 1103                                     | 100                    | 120                           | 1                     |
| libxml2      | 1092                                     | 2                      | 100                           | 1                     |
| libsqlite3   | 1082                                     | 2                      | 100                           | 1                     |
| libjpeg-turbo| 1073                                     | 1                      | 100                           | 1                     |
| libglog      | 1064                                     | 1                      | 100                           | 1                     |
| libgpg        | 1055                                     | 10                     | 10                            | 1                     |
| libpng       | 1038                                     | 100                    | 10                            | 1                     |
| libxml2       | 1026                                     | 2                      | 100                           | 1                     |
| libglog       | 1017                                     | 2                      | 100                           | 1                     |
| libgd         | 1009                                     | 6                      | 16                            | 1                     |
| libglog       | 999                                      | 2                      | 180                           | 2                     |

**TABLE VII:** The top 20 most neglected/outdated native libraries in our dataset (Taking frequency into account)

| Library Name | Avg. Outdatedness Across All Apps (days) | Num. Releases Per Year | Rate of Library Release (days) | No. of Apps Using Lib |
|--------------|------------------------------------------|------------------------|-------------------------------|------------------------|
| libunwind    | 1167                                     | 25                     | 240                           | 2                     |
| libgd         | 1114                                     | 6                      | 180                           | 2                     |
| libopenal    | 1103                                     | 100                    | 120                           | 1                     |
| libxml2      | 1092                                     | 2                      | 100                           | 1                     |
| libsqlite3   | 1082                                     | 2                      | 100                           | 1                     |
| libjpeg-turbo| 1073                                     | 100                    | 10                            | 1                     |
| libglog      | 1064                                     | 100                    | 10                            | 1                     |
| libgpg        | 1055                                     | 100                    | 10                            | 1                     |
| libpng       | 1038                                     | 100                    | 10                            | 1                     |
| libxml2       | 1026                                     | 2                      | 100                           | 1                     |
| libglog       | 1017                                     | 2                      | 100                           | 1                     |
| libgd         | 1009                                     | 6                      | 16                            | 1                     |
| libglog       | 999                                      | 2                      | 180                           | 2                     |
| libgd         | 992                                      | 6                      | 16                            | 1                     |
| libglog       | 981                                      | 2                      | 180                           | 2                     |
| libgd         | 972                                      | 2                      | 180                           | 2                     |
| libgd         | 961                                      | 2                      | 180                           | 2                     |
| libgd         | 950                                      | 2                      | 180                           | 2                     |

| Library Name | Avg. Outdatedness Across All Apps (days) | Num. Releases Per Year | Rate of Library Release (days) | No. of Apps Using Lib |
|--------------|------------------------------------------|------------------------|-------------------------------|------------------------|
| libunwind    | 1167                                     | 25                     | 240                           | 2                     |
| libgd         | 1114                                     | 6                      | 180                           | 2                     |
| libopenal    | 1103                                     | 100                    | 120                           | 1                     |
| libxml2      | 1092                                     | 2                      | 100                           | 1                     |
| libsqlite3   | 1082                                     | 2                      | 100                           | 1                     |
| libjpeg-turbo| 1073                                     | 100                    | 10                            | 1                     |
| libglog      | 1064                                     | 100                    | 10                            | 1                     |
| libgpg        | 1055                                     | 100                    | 10                            | 1                     |
| libpng       | 1038                                     | 100                    | 10                            | 1                     |
| libxml2       | 1026                                     | 2                      | 100                           | 1                     |
| libglog       | 1017                                     | 2                      | 100                           | 1                     |
| libgd         | 1009                                     | 6                      | 16                            | 1                     |
| libglog       | 999                                      | 2                      | 180                           | 2                     |
| libgd         | 992                                      | 6                      | 16                            | 1                     |
| libglog       | 981                                      | 2                      | 180                           | 2                     |
| libgd         | 972                                      | 2                      | 180                           | 2                     |
| libgd         | 961                                      | 2                      | 180                           | 2                     |
| libgd         | 950                                      | 2                      | 180                           | 2                     |
version per year from their official websites. The number of releases per year ranged from a single release in the case of Ogg (libogg), a bitstream codec library, to as many as 37 releases per year in the case of Folly (libfolly), the Facebook Open-source Library. The Unity 3D library (libunity) occurred the most in our app repository, totalling 2,412 out of 12,646 app versions (19%). Unity 3D releases, on average, 13 updates per year (i.e., they release an update every 28 days). In our data set, libunity remained outdated for an average of 105 days, indicating that app developers are 3.75 times slower at updating to new releases of libunity.

libadcolony, libcrashlytics, libgpg and libopenpencv release a new update every 60 days (6 updates per year). However, our results indicate that app developers are 3.39 times slower in terms of including the newly released updates of these libraries.

Developers of libgp甫image, libvpx, and libopenhall release updates less frequently (every 5 months) compared to the rest of the libraries in Table VII. Yet, apps containing instances of these 3 libraries have an average library outdatedness of 10, 14, and 9 months for libgp甫image, libvpx, and libopenhall, respectively.

Libogg releases, on average, a single new version every year—the slowest release rate of all the libraries among the top 20 most outdated libraries in Table VII. However, even with this slow release rate, instances of this library in apps remain unchanged for 1.7 years.

The most neglected library is libcrypto, which remains outdated for an average of 13 months (400.16 days). New versions of libcrypto were released every 36 days. As a result, app developers were 11.12 times slower at updating to new releases of libcrypto. Given the security and privacy implications of not updating this cryptography library, the fact that this library was the most neglected is particularly concerning.

To summarize the overall outdatedness results, Table VII depicts third-party libraries that remain unchanged for an average of 252.12 ± 29.40 days. New releases of these libraries are made available every 83.05 ± 18.49 days—indicating that instances of native libraries in Android apps are often neglected for long periods of time before getting updated. This degree of outdatedness jeopardizes end-user security, especially if these outdated libraries contain security flaws.

OpenCV Case Study. In addition to studying the prevalence of outdated libraries in our app repository, we conduct a case study of the outdatedness of one particular native library, OpenCV, which is a library for real-time computer vision. OpenCV is particularly interesting because it was used by 10 popular apps but no app used more than 3 versions of the library—making it possible to effectively visualize the timeline of OpenCV releases and potential OpenCV updates for those 10 apps. Figure 3 shows that timeline for those 10 popular apps and the time at which three possible versions of OpenCV were used. Metadata for those 10 popular apps is presented in Table VIII. Each of the 10 apps used a maximum of 3 Library Instances of OpenCV, as depicted by three different colored bars, one for each Library Instance, in Figure 3. Each of these instances correspond to a single library version—although, it is possible that all three instances may actually be the same library version.

There were 30 available releases of OpenCV between 2014 and 2019. Figure 3 visualizes releases, in two-month intervals, as broken orange vertical lines. This relatively high frequency of releases for OpenCV give developers of the 10 apps ample opportunity to keep their apps updated with a recent version of OpenCV. Ubercab, for example, used one instance of OpenCV between late 2013 and early 2016 before switching to a second instance that remained in the app until it was finally changed to a third and final instance in mid-2018. During that period (2013-2019), OpenCV released 30 updates, while Ubercab only utilized up to 3 versions (only 11% of total updates). Even with the latest update to OpenCV in Ubercab, there were still at least 3 newer releases that Ubercab could have updated to but did not.

PayPal and GroupOn used 3 different instances during this timeline. Similarly, PayPal used only 3 instances of OpenCV between late 2013 and late 2018 before it was completely removed. GroupOn switched between 3 different instances of OpenCV in 2016 before reverting back to the second instance which remained outdated until early 2019 before switching back to the first instance. As a result, GroupOn had at least 9 newer releases of OpenCV it could have updated to.

OpenCV was included in both Uber Driver and Uber Eats in early/mid 2016 and it remained outdated for two years before they updated to another instance of OpenCV which remained unchanged until the time of app collection. Between that same period (2018 and onward), 10 new releases of OpenCV were made available.

The remaining apps—Lyft, American Airlines, GrubHub
and Eventbrite—use only one instance of OpenCV. The former two apps still used the same outdated instance of OpenCV despite the fact that at least 10 newer releases of OpenCV were available—3 of which were security patches.

C. RQ3: Prevalence of Vulnerable Libraries in Top Apps

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 in Section IV and then apply an additional manual verification step to determine exact version information. Once the versions are identified, we investigate the extent to which native libraries of Android apps are vulnerable and remain vulnerable.

Experiment Setup. Due to the time-consuming nature of manually verifying the exact versions of libraries, we selected 500 Library Instances (i.e., clusters of native binaries obtained using BinSim Clustering), which cover 21,236 binaries (24% of the total binaries). We focus on binaries with libraries that (i) are found in a greater number of apps and (ii) have known CVEs.

We first manually locate source code of libraries for the selected 500 Library Instances. To that end, we use readily available auxiliary data such as keywords found in feature vectors, binary filenames, and dependencies. Once we identify potential source code, 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. For RQ3, we obtained such binaries from official websites, GitHub, and Debian repositories. The binaries with known libraries and versions for RQ3 contain 52 distinct libraries with a total of 961 versions and an average of 12 versions per library. Table XIV shows more detail about these 52 libraries.

Results. We found that, out of 21,236 binaries for which we inferred their versions, 3,614 were vulnerable libraries (17%) affecting 80 distinct apps with a total of 1,781 app versions. 285 app versions (41 distinct apps) released between Feb 2019 and April 2019 include a library with reported CVEs.

Table IX shows the list of native libraries with reported vulnerabilities along with the number of affected apps between the period of Sept. 2013–April 2019. Furthermore, we also report the number of apps versions released between Feb. 2019–April 2019 that include a vulnerable library.

OpenCV and GIFLIB affect the most apps. OpenCV has the largest number of affected apps with a total of 696 versions (42 distinct) where 100 recent app versions (14 distinct) still have a vulnerable instance of OpenCV. Note that 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 594 app versions (23 distinct). GIFLIB continues to affect 105 apps up to the point in time we stopped collecting apps (April 29th, 2019).

SQLite3 has the largest number of vulnerable versions (10 in total) included in 21 apps with a total of 351 app versions. 18 apps released this year contain a vulnerable version of SQLite3 in April 2019.

One vulnerable version of XML2 was found in 38 versions of Microsoft XBox SmartGlass and the library was not updated for 6 years—still remaining vulnerable up to the point where we collected apps for our repository. This particular case is notable due to the extremely long amount of time the library had been vulnerable and remained vulnerable.

Our results show that 80 apps (1,781 versions in total) have been affected by a minimum of 1 vulnerable library and a maximum of 4 vulnerable libraries covering dates between Sept 2013 and April 2019. 61 of those apps still include a vulnerable binary at the time of our app collection.

Table X shows 10 popular apps that are using at least one library with a reported CVE at the time of our app collection. We select this specific set of apps to highlight that vulnerable libraries exist in apps with various installs (10M+) and across different app categories. Moreover, we include apps that were found to have multiple vulnerable libraries at the same time.

TikTok, a social-media video app with a total of more than 500M installs, uses version 5.1.1 of GIFLIB with associated CVE-2015-7555 since April 2016. Similar to TikTok, PicsArt uses an instance of GIFLIB (version 5.1.4) which is vulnerable since Feb 9th, 2019 and continued to be used in the app up to the time of our app collection.

Lyft and Uber, two major ridesharing apps, use OpenCV-2.4.11 and OpenCV-2.4.13, respectively. Not only are these versions outdated—OpenCV-2.4.11 was released in July 2015, and OpenCV-2.4.13 was released in April 2016—they are also known to be vulnerable since August 2017 yet remain unchanged in both apps. Two other subsidiaries of Uber, UberDriver and UberEats, also contain the same vulnerable version of OpenCV.

Kik Messenger, which has a download base of over 100M, contains two vulnerable libraries of OpenSSL-1.0.1s and FFmpeg-2.2 both with a published CVE in Sept. 2016 and Dec. 2016, respectively. Kik continued to use both vulnerable versions until the date of our app collection.

Another application that contains two vulnerable libraries is Amazon Alexa, a virtual assistant used in Amazon Echo smart speakers. It includes OpenSSL-1.0.1s which was reported vulnerable 3 years ago (a few months before the release of Amazon Alexa) and SQLite-3.11.0 which is associated with CVE-2018-20346 since Dec 2018.

We found 80 apps with up to 1,781 vulnerable versions between Sept, 2013 and April 2019. Moreover, 61 apps (totaling 1,464 versions) remain vulnerable even at the time at which we collected apps for this study with an average outdatedness of 628 ± 73.4 days. These results indicate that apps are likely to remain vulnerable even significantly after the time at which we stopped collecting apps.

D. RQ4: Developers Awareness of Vulnerable Libraries

In this section, we investigate developers awareness of vulnerable libraries and the speed at which they apply security
TABLE IX: A list of libraries with reported CVEs found in our repository along with the number of apps that were affected by a vulnerable library and the number of those apps released in 2019 containing a vulnerable version. #Apps is the number of affected apps; #Vers/App is the number of versions per app.

| LibName     | # Vul. LibVers | Vul. LibVers         | # Vers/App (#Apps) | # Vers/App in 2019 (#Apps) |
|-------------|----------------|----------------------|--------------------|---------------------------|
| OpenCV      | 6              | 2.4.2, 2.4.11, 2.4.13, 3.1.0, 3.2.0, 3.4.1 | 696 (42)           | 100 (14)                  |
| WebP        | 3              | 0.3.1, 0.4.2, 0.4.3  | 293 (18)           | 3 (1)                     |
| GIFLIB      | 2              | 5.1.1, 5.1.4        | 594 (23)           | 105 (13)                  |
| AVCodec     | 6              | 54.92.100, 55.18.102, 55.39.101, 55.52.102, 56.1.100, 57.64.101 | 147 (5)           | 18 (2)                     |
| AVFiler     | 1              | 6.65.100            | 31 (1)             | 9 (1)                     |
| AVFormat    | 4              | 55.19.104, 55.33.100, 56.4.101, 57.56.101 | 137 (4)           | 18(2)                     |
| AVUtil      | 4              | 52.18.100, 52.48.101, 52.66.100, 54.20.100 | 122 (4)           | 9 (1)                     |
| swscale     | 4              | 2.2.100, 2.5.101, 2.5.102, 3.0.100 | 124 (4)           | 9 (1)                     |
| SQLite3     | 10             | 3.6.22, 3.7.17, 3.8.1, 3.8.2, 3.8.7.4, 3.8.10.2, 3.9.2, 3.11.0, 3.13.0, 3.14.1 | 351 (21)           | 66 (18)                    |
| XML2        | 1              | 2.7.7               | 38 (1)             | 4 (1)                     |
| OpenSSL-crypto | 8         | 1.0.0a, 1.0.0r, 1.0.1e, 1.0.1c, 1.0.1r, 1.0.1s, 1.0.2f, 1.0.2h | 218(8)           | 25 (3)                     |

Table X: 10 popular apps from Google Play which include a vulnerable library that remained unchanged until our collection of apps in April 2019.

| AppName     | Genre  | Num. Installs | Vulnerable Libs |
|-------------|--------|---------------|-----------------|
| TikTok      | Social | 500M+         | GIFLIB-5.1.1    |
| PixaArt     | Photos | 500M+         | GIFLIB-5.1.4    |
| Uber        | Navigation | 100M+        | OpenCV-2.4.13   |
| Kik         | Communication | 100M+        | FFmpeg-2.2, OpenSSL-1.0.1s |
| eBay        | Shopping | 100M+        | FFmpeg-2.4.13   |
| My Talking Angela | Games | 100M+        | SQLite-3.13.0   |
| Amazon Alexa | Music | 10M+         | OpenSSL-1.0.1s, SQLite-3.11.0 |
| GIPHY       | Editors | 10M          | FFmpeg-2.2      |
| Wells Fargo  | Finance | 10M+        | OpenCV-3.1.0    |
| Lyft        | Navigation | 10M+        | OpenCV-2.4.11   |

policies. To determine the rate at which developers update vulnerable libraries, 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 from Section IV that we collected the previous versions of the top 600 apps from Google Play. Moreover, we inferred the library versions from 21,236 apps 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 19 apps (317 versions in total) with vulnerable versions of FFmpeg, GIFLIB, OpenSSL, WebP, SQLite, and OpenCV between Sept 2013 and April 2019. We exclude apps that removed a library before a CVE was associated with it and apps containing libraries that are vulnerable up to the time of collection. 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 a change was made to the vulnerable library, i.e., either updating to a new version or removing the library.

We found that on average, library developers release a security patch after 19.04 ± 14.35 days from a reported CVE. App developers apply these patches, on average, after 507.21 ± 70.97 days from the date an update was made available—which is about 27 times slower than the rate at which library developers release security patches. This is a concerning difference that exposes end-users to long vulnerability periods, especially considering that library developers released fixed versions much sooner.

Table XI illustrates the slow rate at which app developers applied security patches for vulnerable versions of the following libraries FFmpeg, GIFLIB, OpenSSL, WebP, SQLite, and OpenCV. 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.

A denial-of-service vulnerability was found in versions 1.2, 2.1, and 2.4 of FFmpeg in June 2013, March 2014, and November 2014, respectively. The average number of days a security patch was released for these three vulnerable library versions is 26.67. However, developers took nearly 3 years to address vulnerabilities in Text Me, 2.6 years for Calm, and 2.2 years for InShot. Text Me and Calm opted for library removal, while InShot updated to FFmpeg-2.8.

Facebook and Facebook Messenger, both contained OpenSSL-1.0.1e which was announced as vulnerable in Dec. 2013. OpenSSL developers provided a security patch 14 days after; however, developers of Facebook and Facebook Messenger took 937 days and 801 days, respectively, to remove the vulnerable library. Waze, another app that used a vulnerable version of OpenSSL, removed that vulnerable version on March 2017, 162 days after a security update was released.

A heap-based buffer overflow was reported in GIFLIB-5.1.1 at the end of 2015. The results show that 7 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 610 days, which is 35.88 times slower. This lag time is particularly concerning since GIFLIB released a fix only 17 days after the vulnerable version.

Twitter, GoodRx, Amazon and BIGO include versions 0.4.2 and 0.4.3 of WebP which was fixed for an integer overflow vulnerability in Oct. 2016. However, the apps containing vulnerable WebP versions applied a fix at a much slower pace—with Twitter taking 619 days, GoodRx taking 549 days, and Amazon and BIGO both taking 359 days. BIGO eliminated all native libraries in their app, while the remaining apps only
removed the vulnerable library.

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 affect 5 apps in total. The library was removed from Twitter and Instagram after 431 days, and from Taco Bell after 186. SUBWAY and Wish were the fastest in terms of applying a security patch with an average time-to-fix of 36 days.

Finally, SQLite3 released version 3.26.0, which fixes an integer overflow found in all versions prior to 3.25.3. Waze updated to the fixed version nearly two months after release of the associated security update, while United Airlines removed the library completely 80 days later.

The previous results show that app developers update to new library versions very slowly—even if the existing version contains severe security or privacy vulnerabilities—which further places millions of users at risk, especially when a vulnerability remains unfixed for longer periods of time.

To further understand the consequences of outdated vulnerable libraries, we calculated the average time-to-fix across all vulnerable libraries per app. Table XI lists the top 15 apps with the most number of days a vulnerable library remained in an app until a fix for the vulnerability was applied. Text Me had the longest lag between the vulnerable library being introduced and fixed, i.e., 3 years. SUBWAY was the fastest at almost 2 months. Individual apps had as few as over 5 million installs and as many as over a billion installs.

Social-media apps took years, on average, to fix vulnerable libraries. Facebook, Facebook Messenger and Instagram—which have the largest install base with a total of 1 Billion downloads—fix their vulnerable libraries on average after 690 days. Following Facebook, Inc. apps in terms of the largest number of installs is Twitter with a total of 500M+ installs. Similar to Facebook, Twitter has a slow fix rate of 690 days. With billions of installs, these very long times to fix vulnerable libraries in highly popular social-media apps places users at significant security and privacy risks.

The remaining apps, have an average time-to-fix of 383.35 ± 116.4 days and range in the number of installs between 5M+ and 100M+. With library developers releasing security patches at an average rate of 19.04 ± 14.35 days, app developers are still updating about 20 times slower, leaving their users at substantial risk of being victims to exploits of

| AppName         | Genre          | Installs | AVG. Time-To-Fix (Days) |
|-----------------|----------------|----------|-------------------------|
| Text Me         | Social         | 10M+     | 1148                    |
| Calm            | Health & Fitness | 100M+  | 985                     |
| Facebook        | Social         | 1000M+   | 937                     |
| InShot          | Photography    | 100M+    | 810                     |
| Facebook Messenger | Communication | 1000M+ | 801                     |
| Twitter         | Social         | 500M+    | 690.5                   |
| GoodRx          | Medical        | 5M+      | 620.5                   |
| PayPal          | Finance        | 50M+     | 431                     |
| Amazon Shopping | Shopping       | 100M+    | 430.5                   |
| Instagram       | Social         | 1000M+   | 333                     |
| Taco Bell       | Food & Drink   | 5M+      | 186                     |
| Airbnb          | Travel & Local | 50M+    | 136                     |
| Waze            | Navigation     | 100M+    | 113.5                   |
| United Airlines | Travel & Local | 10M+   | 80                      |
| SUBWAY          | Food & Drink   | 5M+      | 55                      |

TABLE XII: Top 15 most negligent apps in terms of the average time to fix a vulnerable library
the libraries their apps use.

Table XIII lists the top 5 most neglected vulnerable libraries across all apps. FFmpeg is the most neglected app with an average time-to-fix of 2.7 years; WebP is the least neglected library with an average time-to-fix of 1.1 years. Among these 5 libraries, the fact that it takes app developers 633 days, on average, to update or remove vulnerable versions of OpenSSL is particularly concerning due to its security-critical nature. Regardless, all these apps give attackers ample amounts of time to produce exploits for these known vulnerabilities.

VI. DISCUSSION:
Outdated and Vulnerable Native Libraries. For three research questions in Section V we investigated the degree of outdatedness of third-party native libraries in Android apps, the prevalence of vulnerabilities in native libraries, and the extent to which app developers are aware of such vulnerabilities.

Our results from RQ2 (Section V-B) indicate that the top 20 most neglected libraries in our repository (as described in Section IV) have an average outdatedness of 252.12 ± 29.40 days, despite the fact that new releases of these libraries are made available every 83.05 ± 18.49 days. This indicates that instances of native libraries in Android apps are often neglected for long periods of time before getting updated. This degree of outdatedness jeopardizes end-user security, especially if these outdated libraries contain security flaws. At the same time, this slow rate of library updates in Android apps indicate that tracking library dependencies, their associated security updates, and ensuring these new libraries can be included in apps without introducing new regressions is a challenge. One possible avenue of future work addressing this challenge includes constructing means of automating native library updates for Android apps—especially in ways that may reduce possible regressions in them.

Findings in RQ3 (Section V-C) demonstrate that out of 21,236 binaries for which we inferred their versions, 3,614 were vulnerable libraries (17%) affecting 80 distinct apps with a total of 1,781 app versions between Sept 2013 and April 2019. We found 80 apps with up to 1,781 vulnerable versions between Sept. 2013 and April 2019. This constitutes about 13% of the top 600 apps on Google Play. More alarmingly, 61 apps with a total of 1,464 versions remain vulnerable even at the time at which we collected apps for this study with an average outdatedness of 628.7±73.4 days. These results indicate that apps are likely to remain vulnerable even significantly after the time at which we stopped collecting them. 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 RQ4 (Section V-D), 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 19.04 ± 14.35 days from a reported CVE. While app developers apply these patches on average after 507.21 ± 70.97 days from the date an update was made available (27 times slower). Recall that we only consider apps in these cases that actually ended up fixing vulnerable native libraries. The results are even more severe for apps that do not fix those libraries (e.g., 628 ± 73.4 days from RQ3). The results for RQ3 and RQ4 corroborate the need to make app developers aware of the severe risks they are exposing their users to by utilizing vulnerable native libraries. Even for developers that actually fix vulnerable native libraries the fastest, 1.2 years (i.e., 436.24 days) gives attackers plenty of time to create an exploit against their apps.

An interesting finding of our research is that certain apps that are likely to be less security- or privacy-sensitive (e.g., food and drink apps such as Taco Bell and Subway) have much faster average time-to-fix rates than apps that are known to have inherent security or privacy concerns (e.g., social-media apps). Taco Bell’s average time to fix was 3.1 months; Subway was almost 2 months. However, Facebook, Inc., Instagram, and Twitter had a slow fix rate of about 1.89 years. A follow-up study looking into the potential reasons for this stark and surprising difference would be interesting future work.

Overall, our results demonstrate the degree to which native libraries are neglected in terms of keeping them outdated or 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 540 out of 600 top free apps (90%). Interesting future work for our study includes uncovering the root causes of such negligence and means of aiding developers to more quickly update their native libraries (e.g., providing mechanisms to automatically update native libraries while also testing for regressions and possibly automatically repairing them).

Limitations of LibRARIAN. The results from RQ1 shows that LibRARIAN detects versions of native libraries with high accuracy (92.53%). The need to compare against binaries with a known number of versions and libraries (i.e., Known Lib Versions in Figure 2) limits LibRARIAN. Specifically, false negatives reported in RQ1 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 bin²sim that exists in Known Lib Versions. One possibly 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.

Recall from Section II that our feature vectors are built from syntactic symbols such as exported and imported functions, and global variables. Although LibRARIAN reported fewer false positives than OSSPolice (as shown in Sec
we determined that our feature vectors sometimes do not distinguish between 2-3 consecutive minor or patch versions. Potential enhancements that may improve LibRARIAN and allow it to better distinguish between such consecutive minor or patch versions including the following: (1) using Library Instances to aid in determining more fine-grained differences among similar binaries, (2) identifying a higher threshold of similarity specifically for minor or patch versions (e.g., a threshold above 85% that is currently used for Version Identification in Figure 2), or (3) utilizing supervised learning as previously mentioned.

LibRARIAN focuses on benign native libraries that are used as they are—i.e. they are not tampered with by an adversary nor are they modified by app developers or fused with other libraries. A library is fused if a single binary actually contains significant functionality from multiple libraries. Our approach as designed does not identify tampered or fused libraries due to the fact that bin2sim’s use of the Jaccard coefficient penalizes the score for binaries being compared if they do not contain the exact same features. However, from our own experimentation, we have found that replacing the Jaccard coefficient with the overlap coefficient, which checks if one binary $b_1$ has a subset of features of another binary $b_2$ can identify fused or tampered libraries. The main open problem in this regard is determining when to switch from using the Jaccard coefficient to the overlap coefficient. Regardless, we found very few instances of fused or tampered libraries in our data set. Furthermore, we found that the Jaccard coefficient is resilient against internal clones (i.e., third-party library source code that is reused in the source code of another library), which OSSPolice fails to detect.

VII. 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 [12, 9]. Derr et al. [12] 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. [9] 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.

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 [18], [11]; supervised machine learning [23], [21]; and code clustering [27], [22], [20]. LibScout [9] 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 [17][16]. VulSeeker [17] matches binaries with known vulnerabilities using control-flow graphs and machine learning. discovRE [15] operates at the function level and focuses on identifying vulnerabilities even across CPU architectures.

Binary Analysis Tool (BAT) [19] and OSSPolice [14] 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. Neither BAT [19] nor OSSPolice [14] can detect internal code clones, while LibRARIAN can, giving it superior ability to identify versions of native libraries. Furthermore, BAT and OSSPolice rely on simple syntactical features (e.g., string literals and exported functions). Our feature vectors extract additional features—such as imported functions, exported and imported global variables, and dependencies that uniquely identify different versions of binaries. As shown in Section VII-A2, these additional features were a major factor in the superior accuracy of LibRARIAN compared to OSSPolice.

None of this aforementioned related work has examined the outdatedness of native libraries in Android apps and the prevalence and 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.

VIII. CONCLUSION

Third-party libraries have become ubiquitous among popular apps in the official Android market, Google Play, with 540 out of the 600 top free apps on Google Play (90%) 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 number of attacks if security vulnerabilities remain unfixed.

To determine the extent to which these native libraries remain outdated or vulnerable in Android apps, we study the prevalence of native libraries in the top 600 apps on Google Play across 12,646 versions of those apps. From these versions, we extracted 89,525 native libraries. To identify versions of apps, we constructed on approach called LibRARIAN that leverages a novel similarity metric, bin2sim, that is capable of identifying versions of native apps with a high accuracy—a 92.53% true-positive rate, no false positives, and a 7.46% false-negative rate.

Using LibRARIAN, we determine that native libraries in apps are updated, on average, 3 times slower than the release rate of new versions of those libraries. For vulnerabilities, we found 80 apps with 1,781 vulnerable versions with known CVEs between Sept. 2013 and April 2019, with 61 of those apps still remaining vulnerable until the end point of our study. We find that app developers took, on average, 507.21 ± 70.97 days to apply security patches, while library developers release a security patch after 19.04 ± 14.35 days—a 27 times slower rate of update.

REFERENCES

[1] 440 million android users installed apps with an aggressive advertising plugin. https://www.zdnet.com/article/440-million-android-users-installed-apps-with-an-aggressive-advertising-plugin/

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

[3] gnu.org. https://developer.android.com/ndk
In Proceedings of the 38th International Conference on Software Engineering Companion - ICSE ’16, pages 653–656. ACM Press.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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 - ISTA 2015, pages 71–82. ACM Press.

[28] 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. page 13.

IX. APPENDIX

Table XIV lists the 52 libraries used as ground truth by LibKARIAN for the identification and detection of unknown libraries (Section IV).
| Library Name  | # Library Versions |
|--------------|---------------------|
| AdColony     | 24                  |
| AVCodec      | 18                  |
| AVFilter     | 25                  |
| AVFormat     | 36                  |
| AVUtil       | 27                  |
| Breakpad     | 13                  |
| Card.io      | 30                  |
| Cocos2d      | 25                  |
| Crashlytics  | 16                  |
| Cronet       | 12                  |
| EXIF Library | 24                  |
| Facebook folly | 25              |
| Facebook fresco | 49            |
| Firebase     | 32                  |
| GFFLIB       | 20                  |
| GLib         | 25                  |
| JavaScriptCore | 14             |
| Libglog      | 14                  |
| Libgpg       | 23                  |
| Liblconv     | 10                  |
| Libjpeg      | 7                   |
| Libncurses   | 10                  |
| Libopus      | 10                  |
| libpng       | 11                  |
| Libsepol     | 10                  |
| Libnet       | 11                  |
| Libunrar     | 9                   |
| Libunwind    | 9                   |
| libvpx       | 24                  |
| LibZ2        | 6                   |
| MapBox GL    | 6                   |
| Mono         | 24                  |
| mpg123       | 12                  |
| MupDF        | 10                  |
| OGG          | 6                   |
| OpenAL       | 16                  |
| OpenCV       | 112                 |
| OpenSSL      | 14                  |
| RenderScript | 6                   |
| SDL          | 16                  |
| SDL_image    | 15                  |
| SDL_mixer    | 13                  |
| SDL_net      | 12                  |
| SDL_Pango    | 14                  |
| SDL_tiff     | 10                  |
| SQLite       | 14                  |
| swresample   | 11                  |
| swscale      | 13                  |
| Vorbis       | 15                  |
| WebP         | 41                  |
| XML2         | 12                  |

TABLE XIV: Libraries used as ground truth for our study