Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android

Erik Derr, Sven Bugiel
CISPA, Saarland University
Saarland Informatics Campus

Sascha Fahl, Yasemin Acar
Leibniz University Hannover

Michael Backes
CISPA, Saarland University
Saarland Informatics Campus

ABSTRACT

Third-party libraries in Android apps have repeatedly been shown to be hazards to the users' privacy and an amplification of their host apps' attack surface. A particularly aggravating factor to this situation is that the libraries' version included in apps are very often outdated.

This paper makes the first contribution towards solving the problem of library outdatedness on Android. First, we conduct a survey with 203 app developers from Google Play to retrieve first-hand information about their usage of libraries and requirements for more effective library updates. With a subsequent study of library providers' semantic versioning practices, we uncover that those providers are likely a contributing factor to the app developers' abstinence from library updates in order to avoid ostensible re-integration efforts and version incompatibilities. Further, we conduct a large-scale library updatability analysis of 1,264,118 apps to show that, based on the library API usage, 85.6% of the libraries could be upgraded by at least one version without modifying the app code, 48.2% even to the latest version. Particularly alarming are our findings that 97.8% out of 16,837 actively used library versions with a known security vulnerability could be easily fixed through a drop-in replacement of the vulnerable library with the fixed version. Based on these results, we conclude with a thorough discussion of solutions and actionable items for different actors in the app ecosystem to effectively remedy this situation.

1 INTRODUCTION

Third-party libraries are an indispensable aspect of modern software development. They ease the developer's job, for instance, by providing commonly useful functionality, sharing programming know-how among developers, enabling monetization of software, or integrating social media such as Facebook or Twitter. In contrast to the benefits that developers reap from third-party code, end-users of software are reportedly exposed to an increasing risk to their privacy and security by those external software components. Recent reports [17, 38] warn of the hidden costs of libraries in form of buggy code that increases the app's attack surface and introduces security vulnerabilities. Sonatype [38] reports that older software components have a three times higher rate of vulnerabilities and that almost 2 bn software component downloads per year include at least one security vulnerability. These numbers are backed with findings from different software ecosystems, e.g., for Windows applications [31] and Javascript libraries [23]. Moreover, their results show that, although library updates with security fixes exist, they are not adopted by developers.

Similarly, recent works [4, 9] have reported such alarming findings for the Android ecosystem. About 70% of all third-party libraries in apps are (severely) outdated and a slow adoption rate of updates of about one year aggravates the library outdatedness problem. As a consequence, fast response times by library developers remain noneffective and even known security vulnerabilities [3, 7, 33–35] remain a persistent threat in the app ecosystem, when app developers do not integrate the existing fixes into their apps. Google recognized their central role as market operator early for amending this problem and introduced their application security improvement program (ASI) [12] in 2015. In this ongoing effort, Google notifies developers when security problems were detected in their apps and/or included third-party components and enforces a remediation period to fix the detected vulnerabilities. According to their statistics [14], this approach already proved to be successful in improving the overall app market security. However the main drawback is that this approach only fights the symptoms of the underlying problem of developers not keeping dependencies up-to-date.

To improve on this situation more sustainably, for instance by realizing effective solutions that are practical and accepted by all involved parties, it is important to first understand the app developers' motivation for not updating third-party dependencies and to investigate the role of other actors—like the library developers—in the current situation. This paper makes the first contribution towards such a solution by identifying the root causes *why* app developers do not update third-party libraries on Android. We start with conducting a survey with 203 app developers from Google Play to collect first-hand information about library usage in apps. Among others, this survey covers questions regarding library selection criteria, developer tools, reasons to (not) update, as well as feedback and comments on what app developers think needs to be changed to enable more effective library upgrades. These insights motivate a follow-up library release analysis that uncovers that library developers are very likely a contributing factor to the poor adaptation rate through an inconsistent and imprecise library version specification, i.e., the actual changes in code and API do not match the expected changes conveyed by the version numbers (*semantic versioning*). As a result, app developers cannot properly assess the expected effort for upgrading the library and

CCS '17, October 30-November 3, 2017, Dallas, TX, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134059

ultimately abstain from an update to prevent ostensible effort and incompatibilities.

To investigate the actual effort of updating libraries, we conduct a large-scale library updatability analysis of 1,264,118 apps from Google Play. We analyzed the apps' bytecode to check whether included libraries are actually called by the app. Combining this data with the results of an analysis of each library's API robustness across its different versions, we determine that 85.6% of all libraries can be updated by at least one version, in 48.2% of all cases even to the most current library version, simply by replacing the library and without the need to change the host app's code. Contributing factors for this high updatability rate are a generally low library API usage, i.e., on average 18 library API calls, and the fact that the most frequently used APIs remain stable for the majority of libraries. Focusing on security incidents, we find 16,837 actively used libraries in apps that contain one publicly known security vulnerability. Based on our analysis, 97.8% of these libraries could be patched by simply exchanging the vulnerable library with the fixed version, again without the need to change the app's code.

Finally, the results of the developer survey and our follow-up analyses helped us to identify problem areas and weak links in the ecosystem. In Section 5 we summarize our findings and propose actionable items for different entities including library developers, the market place, development tools, and the Android system to remedy the situation. Based on our findings and the responses from our survey, we believe that these solutions are both effective in amending the library outdatedness problem and accepted by the majority of developers. In summary, this paper makes the following contributions:

(1) We conduct a survey with 203 app developers from Google Play to collect first-hand information on library usage and to identify root causes of developers not updating their dependencies.

(2) We analyze library releases to uncover that library developers are likely a contributing factor to a poor library adaptation. In 58% of all library updates, the expected changes derived from semantic versioning do not match the actual library code changes.

(3) We conduct a large-scale analysis of 1,264,118 apps to identify libraries and their API usage. In 85.6% of cases, the detected library can be updated by at least one version, in 48.2% of cases even to the most current version. In addition, we find 16,837 apps that include a library with a known security vulnerability, out of which 97.8% could be patched without app code adaption.

(4) Finally, we thoroughly discuss short-/long-term actionable items for different entities of the app ecosystem to remedy the problem of outdated libraries.

The remainder of this paper is structured as follows: In Section 2, we explain how we conducted our developer survey and summarize first results. We present the results of our library release analysis in Section 3 and the results of our library updatability analysis in Section 4. We thoroughly discuss our survey and analysis results and propose actionable items in Section 5. We compare with related work in Section 6 and conclude the paper in Section 7.

Table 1: Demographics of developer survey participants.

Gender		**Age ($\bar{x}$ = 32.90 ± 1.60 years)**	
Female	10 (04.93%)	15–19	6 (02.96%)
Male	186 (91.63%)	19–29	63 (31.03%)
No answer	7 (03.45%)	29–39	64 (31.53%)
		39–49	31 (15.27%)
Highest educational degree		49–59	15 (07.39%)
Graduate	117 (57.64%)	59–69	4 (01.97%)
College	41 (20.20%)	No answer	20 (09.85%)
High school	30 (14.78%)		
No degree	12 (05.91%)		
No answer	3 (01.48%)		

2 APP DEVELOPER SURVEY

We conducted an online survey with Android application developers who already published at least one application on Google Play. We investigated the developers' main motives and knowledge when it comes to managing third-party libraries for their apps. Mainly, we were interested in the following three questions:

Q1: What is the common workflow to search for and to integrate third-party libraries into applications?

Q2: How frequently do developers update their apps/libs and what is their main motivation for updates?

Q3: What are possible reasons to not update dependencies and what solutions could app developers think of?

2.1 Ethical Concerns

The questionnaire (see Appendix A) was approved by the ethical review board of our university. We also took the strict German data and privacy protection laws into account for collecting, processing, and storing participant information. We collected email addresses from Android application developers who had previously published at least one application on Google Play and kindly asked them to participate in our online questionnaire, whether they like to be blacklisted for future user studies, and whether they want to learn more about our scientific work. Overall, we sent out 60,000 invite emails. Before filling out the questionnaire, developers had to consent to the use and publication of their answers.

2.2 Participants

In response to the invitation emails, 203 app developers finished the questionaire within five days (participation rate of 0.34%). Of all participants, 91.6% reported being male, 4.9% female, and the remaining 3.4% declined to answer. Participants' mean age was 32.9 years (with a margin of error of 1.6 years with $\alpha = .05$). The general coding experience was relatively high with a mean of 12.11 ± 1.35 years. The Android experience was reported with 4.06 ± 0.33 years on average. Of all participants, 34% affirmed that developing apps is their primary job. Asked about the context of app development, 35.5% reported to develop apps in a company, 38.4% are self-employed, and 61.6% develop apps (also) as a hobby. The participants reported to have worked on 13.188 ± 4.42 apps. A detailed overview of the participants' demographics and professional background can be found in Table 1 and Table 2.

Table 2: Professional background of participants in our online app developer survey.

Years of general coding experience	
< 1 year	4 (01.97%)
1–5 years	45 (22.17%)
5–10 years	51 (25.12%)
10+ years	103 (50.74%)
$\overline{x} = 12.11 \pm 1.35$ years	
Years of Android experience	
< 1 year	2 (01.0%)
1–2 years	19 (09.5%)
2–3 years	28 (14.0%)
4–5 years	40 (20.0%)
5–10 years	37 (18.5%)
10+ years	4 (02.0%)
$\overline{x} = 4.06 \pm 0.33$ years	
How learned Android programming†	
Self-taught	182 (89.66%)
On the job	66 (32.51%)
Online coding course	38 (18.72%)
Class in university	25 (12.32%)
Class in school	8 (03.94%)
Other	0 (00.00%)
Developing apps primary job	
Yes	69 (33.99%)
No	134 (66.01%)
Context of app development†	
Company	72 (35.47%)
Self-employed	78 (38.42%)
Hobby	125 (61.58%)
Company size	
< 10 employees	27 (37.50%)
10–50 employees	16 (22.22%)
50–100 employees	7 (09.72%)
100+ employees	22 (30.56%)
Number of apps worked on	
1–5 apps	101 (50.00%)
6–10 apps	50 (24.75%)
11–50 apps	44 (21.78%)
51–100 apps	5 (02.48%)
100+ apps	2 (00.99%)
$\overline{x} = 13.188 \pm 4.42$ apps	

† Multiple choice, sum does not need to equal 100%

2.3 Q1: Workflow and Integration

In the first part of the survey we seek to answer how app developers choose and integrate libraries into their apps. Figure 1 shows the primary sources of the participants to search for libraries. It is evident that the majority of app developers use search engines, followed by the project hoster GitHub. The relatively small number of dedicated Android community websites, such as *Android Arsenal* or *Android Weekly*, underlines the lack of a central library marketplace/package manager such as *Cocoapods* for iOS or *npm* for JavaScript. Being asked about library selection criteria (see Figure 2), 79.7% of all participants named functionality as main criteria. Open source (61.7%) and good documentation (52.3%) are further criteria for library selection. In general, recommendations and user ratings are less important. Security (26.6%) and particularly the use of permissions (29.7%) are among the least important criteria, which is particularly surprising after news reports and scientific research on permission misuse of advertisement libraries [8, 19, 35, 39].

Besides information about how libraries are chosen, it is important to know the preferred development platform and integration

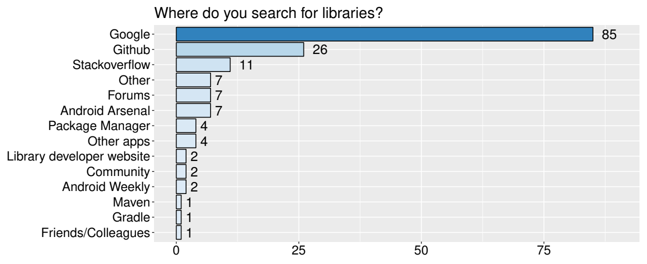

Figure 1: Primary sources for finding libraries among our survey participants

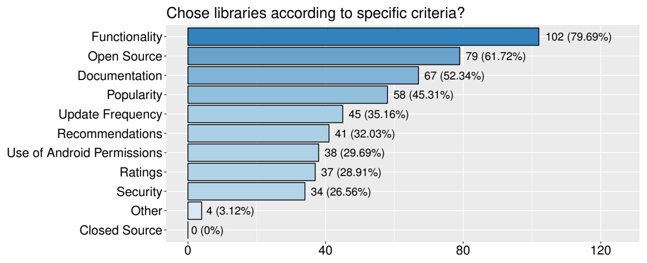

Figure 2: Reported criteria for library selection among our survey participants

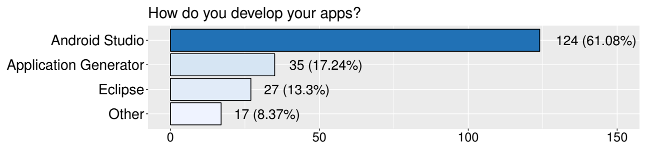

Figure 3: Primary development environment of our survey participants

approach by developers. Figure 3 suggests that *Android Studio* is the preferred IDE for app development (61%), followed by application generator frameworks such as *Xamarin* or *Cordova* (17.2%) and *Eclipse* with the Android plugin (13.3%). A small fraction of app developers (8.4%) prefers different environments such as *NetBeans* or even the command line. Similar to development platforms, there are different possibilities to integrate a library (see Figure 4). The Android *Gradle* plugin, introduced in 2014, is a powerful dependency manager and the default in Android Studio. Although two thirds of app developers use Gradle, more than half of them also resort to manual inclusion or use a combination of different approaches. Build systems such as *Maven* (14%) or *Ant* (3.9%) are not widespread in Android app development. Users of *Xamarin* prefer to use its convenient package manager *NuGet*.

2.4 Q2: Application and Library Maintenance

In the second part of the survey, we asked the participants about app release frequency, whether they update their dependencies, and about their main motivation to perform app and library updates (see Figure 5). 78% of the app developers release new app updates on a variable schedule, while only 22% rely on a fixed schedule,

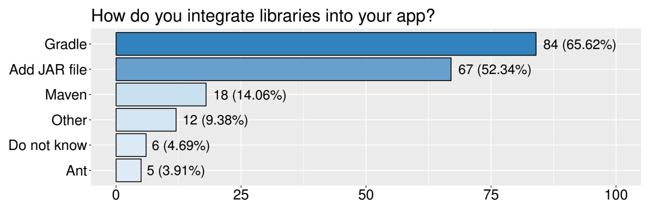

Figure 4: Used library integration techniques by our survey participants

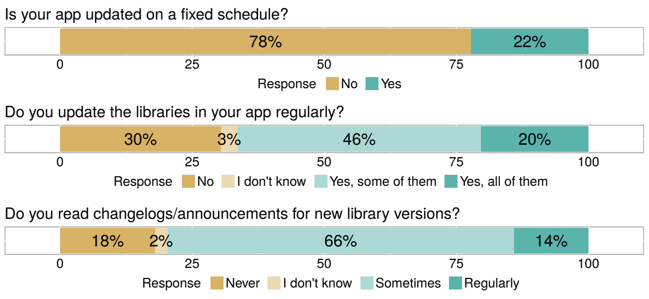

Figure 5: Questions and responses for Q2 regarding app/library release frequency.

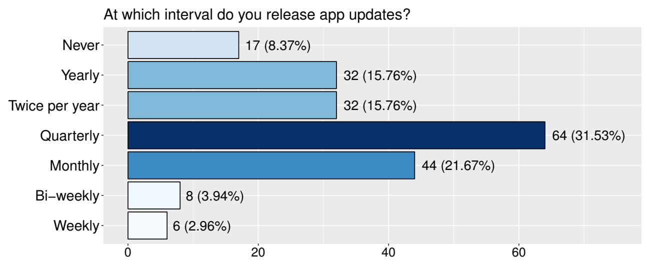

Figure 6: Interval at which our participants release their app updates

e.g., app developers at companies with a fixed release schedule. The majority of developers releases new updates within a time period of one to three months. However, there is also a considerable number of developers (39.9%) that provides updates at most twice a year.

The main motivation to release new app versions is to provide new functionality and fixing bugs (see Figure 7). Only one third of the developers explicitly names library updates as a reason to provide a new app version. This is contrary to the main motivation to update the apps' libraries where the dominant answer is bug fixing (only three developers did not name this). Functionality is only the third most common reason (56.5%), right behind security fixes (57.6%). Of all app developers, 66% update at least some of their libs regularly, while 30% completely abstain from updating the dependencies. Changelogs and release announcements are an effective means to reach app developers, since 70% of the developers read them at least sporadically.

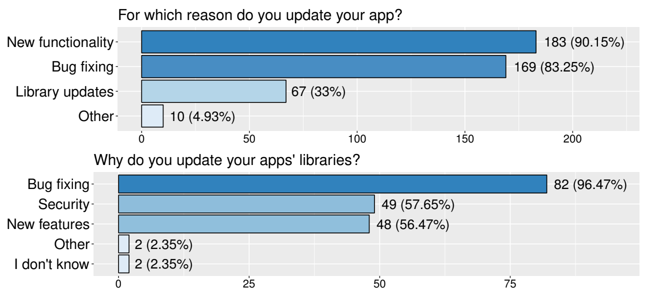

Figure 7: Reasons why our survey participants update apps and their apps' libraries

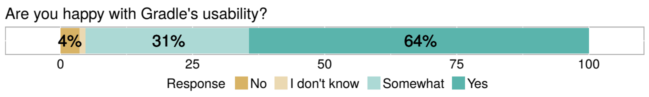

Figure 8: Usability satisfaction of our participants with the Gradle build system

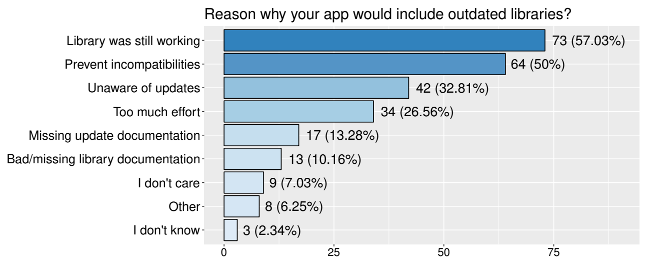

Figure 9: Self-reported reasons why the participants' apps would include an outdated library

2.5 Q3: Reasons for Outdated Libs

The last part of the survey asked questions about problems that might be a reason for not updating libraries. We also requested a self-reporting on reasons for outdated libraries and asked the developers for their opinion on possible solutions.

Since Gradle is the default dependency management system in Android, we asked about Gradle's usability and drawbacks. While the majority of the participants likes Gradle (65.3% in Figure 8) or only sees minor limitations (31%), only three participants are unhappy with Gradle's usability. The most frequently named drawbacks include a weak build performance with more complex apps and a steep learning curve compared to the simplicity of adding libraries manually. We then explicitly asked for reasons that their apps contain outdated libraries (see Figure 9). For 57% of the participants there is no incentive to update the library as it works as intended. Half of the participants are afraid of experiencing incompatibilities, for instance, through modified or renamed library APIs, or they refrain from updating due to an expected high integration effort. Another reason is that app developers are just unaware of library updates (33%).

Figure 10 shows a selection of potential approaches to facilitate better library management. Of all participants, 65.6% wish to have

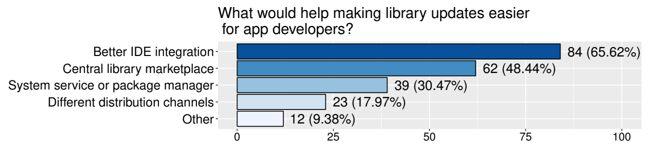

Figure 10: Preferred improvements for making library updates easier

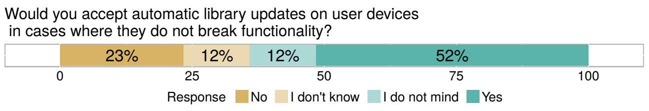

Figure 11: Acceptance of automatic library updates on end-user devices among our participants

better development tools, for instance, an improved IDE integration. Among the app developers, 78.9% like the idea of having a central library market place or package manager, similar as in other ecosystems, such as iOS or JavaScript. Many library developers distribute their libraries via different channels, such as *Maven Central* or *Bintray*. For those who host their library only on their website, developers would welcome additional, potentially more convenient, distribution channels.

Finally, we asked whether participants would accept an automated on-device library patching via the Android OS, as long as it would not break app functionality. Half of the responses fully agreed with such a solution, while about 12% were not sure whether this is a good idea. About 23% clearly disagreed with such an approach, while another 12% did not mind.

2.6 Limitations

As with any user study, our results should be interpreted in context. We chose an online study because it is difficult to recruit Google Play developers for an in-person study at a reasonable cost. Choosing to conduct an online study gave us less control over the recruitment process; however, it allowed us to recruit a large and geographically diverse sample. Because we targeted Google Play developers, we could not easily take advantage of services like Amazon's Mechanical Turk or survey sampling firms. Managing online study payments outside such infrastructures is very challenging; as a result, we did not offer compensation and instead asked participants to generously donate their time. As might be expected, the combination of unsolicited recruitment emails and no compensation led to a strong self-selection effect, and we expect that our results represent Android developers who are interested and motivated enough to participate.

In any online study, some participants may not provide full effort, or may answer haphazardly. In this case, the lack of compensation reduces the motivation to answer in a constructive manner; those who are not motivated will typically not participate in the first place. We attempt to remove any obviously low-quality data (e.g., responses that are entirely invective) before analysis, but we cannot discriminate perfectly.

3 LIBRARY RELEASE ANALYSIS

The survey results indicate that 77% of app developers update at most a strict subset of their included libraries (see Figure 5). One of the main reasons for this is that there is no obvious need to update the library when it works as intended. The survey suggests that bugfixes and security fixes would be a reason to update if new library versions would provide dedicated patch-only changes and would not mix bugfixes with new functionality. Another more alarming reason is that libraries are not updated due to the fear of experiencing incompatibilities and an expected high integration effort. This raises the question *how* library developers release new versions and whether their current release strategy could be a contributing factor to poor library adoption. In the following, we seek to answer this question by analyzing how often library versions change existing APIs and provide versions with mixed types of changes, i.e., security fixes and new functionality. A related but previously uncovered aspect is how library developers communicate these changes, i.e., which changes might an app developer expect given a library version number and do these expectations match the actual changes made in code and API.

3.1 Semantic Versioning

The concept of classifying a version number into different categories to infer the expected effort of integration was proposed as *Semantic Versioning* (SemVer) by Preston-Werner [36]. It comprises a set of simple rules that dictate how *library* developers assign and increment new version numbers. The basic idea is that if library developers adhere to these rules, the library consumer (typically the app developer) can assess, just by looking at the version string, whether or not a library update can be performed without additional implementation and code adaption effort. *Semantic Versioning* works as follows: First, the lib developer declares the public API, e.g., by documenting it. Then, any changes in the documented public API are communicated with the version number. The version format consists of three numbers `X.Y.Z` (`Major.Minor.Patch`). Whenever a new version includes bug fixes or code-only changes that do not affect the API, the patch version number is incremented. Backwards compatible API additions/changes increment the minor version and backwards incompatible API changes (removed methods, incompatible argument types) increase the major number. Intuitively, a library without further dependencies can be updated without additional effort if a new version is a minor/patch version. A major version might require additional integration effort, depending on the changes of APIs in use.

3.2 Android Library Versioning

To investigate the status quo in Android library versioning we conduct an empirical study of expected changes versus actual changes to confirm or disprove that library developers can be a contributing factor to the problem of a poor library adaptation in the Android app ecosystem. To this end, we build on and extend the library database of the *LibScout* project [4]. In total, we analyze 89 distinct libraries with 1,971 versions with a minimum set of 10 versions per library. In our test set all libraries make use of the `X.Y.Z` versioning scheme, except *OrmLite* which uses an `X.Y` scheme. In addition, *Dropbox* (v2.0.5.1) and *FasterXML-Jackson* (v2.4.1.1) include a single library version with a sub-patch level. However, due to the absence of a changelog for these versions, we can not properly assess the necessity of such version numbers. In the following, we describe in more detail how we determine the actual changes in code and the expected changes conveyed by the version number.

Expected Changes. We extend the *LibScout* tool and integrate a version parser that classifies version changes expressed by the version string into `patch`, `minor`, and `major` releases. By comparing consecutive library versions we then retrieve a list of expected changes, e.g., a version 2.4.1 immediately following version 2.3.7 is classified as `minor` release.

Actual Changes. Semantic Versioning requires that the public library API has to be properly defined at some point, either via an explicit documentation or via the code itself. Since some libraries either lack a full documentation or do not provide a history of their API reference, we programmatically extract the public API from the original library SDKs. The public API set of the first version of each library in our dataset is used as a baseline.

1. Filtering undocumented APIs: Undocumented public methods are not meant to be used by an app developer and hence should not be considered part of the public API. By extracting the public API programmatically, we have to filter such methods in a best effort approach (see also Section 5). To this end, we exclude public methods that reside in subpackages named `internal`. Moreover, we conservatively filter classes (and their declared methods) that have been renamed and shortened through an obfuscation tool like ProGuard [20]. Concretely, we consider classes named with one or two lowercase, alpha characters as obfuscated (following ProGuard's renaming rules).

2. Determining actual changes: To determine actual changes between consecutive library versions, we implement an API diff algorithm that operates on two sets of public APIs api_{old} and api_{new}, where api_{old} is the API set of the immediate predecessor version of api_{new}. An API is described by its signature that includes package and class name as well as the list of argument and return type, e.g. `com.facebook.Session.getAccessToken()java.lang.String`. If $api_{old} = api_{new}$ we have a *patch*-level release, i.e., there are code changes only. If $api_{old} \subsetneq api_{new}$, new APIs were added but existing ones did not change. This is classified as a backwards-compatible *minor* release. Whenever api_{old} includes APIs that are not included in api_{new} we conduct a type analysis to check for compatible counterparts in api_{new}. Compatible changes include generalization of argument types, e.g., an argument with type `ArrayList` is replaced by its super type `List`. Generalization on return types is generally not compatible and depends on the actual app code that uses the return value. Since we do not conduct a code analysis we treat non-matching return types as incompatibility. To not suffer from false positives, we furthermore abstain from searching for alternative candidates when the class and/or package name do no longer match, since this may result in ambiguity.[1] Hence we report conservative numbers when searching for API alternatives. If we are able to identify alternatives for all APIs that do not match exactly, we may classify the release as `patch` or `minor`. All other cases are classified as `major` release.

[1]Updating imports is typically done automatically by an IDE like Android Studio and is therefore not considered as incompatibility.

Table 3: SemVer misclassification by type (expected vs. actual change). Highlighted cells are critical as the actual semantic versioning suggests compatibility although the opposite is the case.

		Expected		
		patch	**minor**	**major**
Actual	**patch**	—	5.7%	0.85%
	minor	11.93%	—	0.48%
	major	**15.02%**	**24.02%**	—

3.3 Semantic Versioning Statistics

Applied to our library set, we found that in 58% of all version changes, the library developer incorrectly specified the new version string, i.e., according to Semantic Versioning rules the expected release level did not match the actual release level. Even worse, there is no single library that achieved a 100% correctness in versioning. Only 3/89 libraries (3.4%) correctly classified the release level in more than 80% of all releases, with the *android-oauth-client* library ranking first (93.8%). On the other hand, 10/89 (11.2%) of libraries specified the version string correctly in less than 20% of all cases. Two libraries (*universal-image-loader* and *log4j*) have not specified a single version change correctly. Further, we could not find a positive or negative, statistically significant, correlation between library category (e.g., Advertising, Utilities) and the Semantic Versioning classification score.

A mismatch between expected and actual changes is always disadvantageously for the library consumer in that she can not properly assess whether a new library version can be used as a drop-in-replacement or whether a considerable amount of work has to be spent to integrate the update. The severity of the mismatch, however, depends on the type of inconsistency. In particular, two types of inconsistencies are problematic: if either patch or minor release is expected, but the actual changes indicate a major release (highlighted cells in Table 3). These numbers show that library developers under-specify changes in 39% of all cases, i.e., the version increment is too moderate and suggests compatibility although API changes might break existing applications. In about 6.5% of cases library developers over-specify changes which does not effect compatibility but might impede wide-spread adaptation due to a high expected integration effort.

Figure 12 summarizes the total number of expected and actual changes between consecutive versions by release level for the 1,971 analyzed versions. The expected changes denote how library developers specified new version strings. This distribution is what you would expect for a typical library lifecycle; a stable base API with occasional additions and code-only changes such as bug and security fixes for the majority of releases. However, the reality looks different: 44% of all versions in our analysis were classified as major release due to non-compatible changes and/or removals of existing

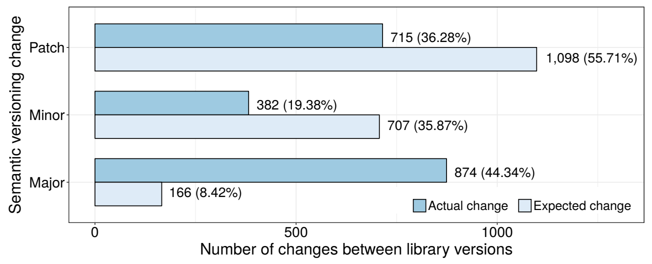

Figure 12: Total number of expected and actual changes between consecutive library versions grouped by patch/minor/major.

APIs. This indicates a poor library design without carefully taking into account the effort/incompatibilities that consumers might experience.

3.4 Security Fixes

Finally, we have a dedicated look at security fixes in libraries. These are the most important kind of updates and should typically be provided as a patch release. However, even when released with a short bug-fixing time, such patches miss their intended effect if they are slowly adapted by app developers or not at all. Ultimately, the end-user will be at risk and suffer from vulnerabilities like identity theft or private data leakage. To check if library developers adhere to this rule, we analyze the *Facebook* and *Dropbox* vulnerabilities used in [4], vulnerabilities in *Apache Commons Collections* (Apache CC) and *OkHttp* found via blog entries, as well as known library vulnerabilities reported by Google's ASI program [12]. In total, we were able to investigate eight distinct bugfix versions[2]. For the eight vulnerable libraries, we first determine whether the bugfix version is a patch release or whether the library provider mixed bugfixes with new content or even changed existing APIs. We subsequently compare these findings with the official changelog to see whether the fix is mentioned and properly documented. Table 4 shows the detailed results.

Six out of ten patched libraries (including two backports) are minor releases, i.e., the developer did not intend to provide a dedicated bugfix version. Only *Airpush* and *Dropbox* provide a patch-level fix, while *Facebook*, *MoPub*, *Supersonic* and *Vungle* provide a major version, i.e., they include new functionality and/or break existing APIs. *Apache CC* and *OkHttp* provide an additional backport of the security patch to allow an effortless adaption by older versions. Surprisingly, both backport versions are patch-only updates, while the fixes for the current releases were announced as minor versions and even included major changes. Mixing critical security patches with API changes is considered bad practice and certainly contributes to a poor adaptation rate. Besides the version number, the changelog is the primary way to convey and explain important fixes and changes to the library consumer (see Figure 5). However, only *Apache CC* explicitly mentions a security fix in its changelog, four libraries at least mention a bug fix. Only the *Dropbox* and *OkHttp* vulnerabilities have a CVE entry. In order to provide transparency and increase the chance that the patch is adapted by developers, some libraries provide a blog/support entry in which they provide additional details about the vulnerability. *MoPub* at least provides a short note in its GitHub repository, referencing the respective ASI support document.

Although we cannot provide the same detailed analysis for the native libraries listed in the ASI program (*libjpeg-turbo*, *libpng*, *libupnp*, *OpenSSL*, and *Vitamio*), we checked their expected SemVer and changelogs for the fix versions. Only *Vitamio* provided the security fix as part of a major release (5.0). All other libraries provide a patch-level version and, more interestingly, even provide detailed changelogs for every (security) bugfix made. In our database of Java/Android libraries, only the Android support libraries and *OkHttp* provide comparable changelogs regarding the level of detail.

4 LIBRARY UPDATABILITY

Keeping third-party dependencies up-to-date is a complex problem with many facets and different parties involved. On the one hand, there are app developers who mainly wish to update libraries for bugfixes and security fixes (see Section 2). On the other hand, there are library developers that want app developers to adapt new library versions within a reasonable time-frame, e.g., for fixes and/or new functionality. In Section 3 we showed that library developers contribute to the adaptation problem by not giving app developers a simple means of assessing whether or not a new library version can be integrated without compatibility issues.

To properly assess the current status quo in library updatability, we have to analyze which library versions and which parts thereof are in use by applications. Given this information, we can then determine whether an actual major library release indeed requires additional integration effort or could still be updated as the set of used APIs remains compatible. To this end we scan 1,264,118 apps from Google Play and identify included library versions. For each found library, we subsequently analyze the application bytecode to determine how the library is used in terms of API calls. Based on that information we infer the highest library version that is fully API compatible for that app/library combination.[3]

4.1 Approach

We base our approach on the open-source project *LibScout* [4]. We extend its implementation to conduct the following analyses:

1. Library API robustness: We first analyze the robustness of the public library API across versions of a given library. For each library with more than 10 versions, we determine, on a per-API level, the highest version that provides this exact API. We are conservative in that we do not search for alternative candidates if the API in question is no longer available, e.g., due to method removal or renaming. Similar to the SemVer analysis, we filter methods that are obfuscated or reside in internal packages. As a result, we receive a comprehensive data set with updatability information for each library version/API pair. Note, that this analysis is much more fine-grained than the API compatibility analysis conducted in [4], which checks whether or not the entire API set of some version is

[2]We had to exclude further vulnerabilities reported by ASI since we were not able to retrieve the original SDKs for either the fixed version and/or some older versions.

[3]There might be cases in which a fully API-compatible library version still breaks the client application. This is discussed in Section 5.3

Table 4: Library versions with a fixed security vulnerability, the expected and actual SemVer, whether and how the security fix is described and whether this library vulnerability is listed in Google's ASI program. Versions marked with (B) denote backport patches.

Library	Fix Version	exp. SemVer	actual SemVer	Changelog	CVE	other	in ASI
Airpush	8.1x	minor	patch	–	–	–	✓
Apache CC	3.2.2 (B) / 4.1	patch / minor	patch /**major**	security	–	blog+report	–
Dropbox	1.6.2	patch	patch	bugfix	CVE-2014-8889	blog	–
Facebook	3.16	minor	**major**	bugfix	–	–	–
OkHttp	2.7.5 (B) / 3.2.0	patch / minor	patch / **major**	bugfix	CVE-2016-2402	blog	–
MoPub	4.4.0	minor	**major**	bugfix	–	GitHub	✓
Supersonic	6.3.5	patch	**major**	–	–	–	✓
Vungle	3.3.0	minor	**major**	–	–	blog	✓

present in the successor version. This data would be insufficient to determine whether an actively used library can be updated.

2. Library usage: To identify the actively used parts of a library, we scan the application bytecode for invocations of this library. To account for identifier renaming obfuscation, we match the library API with the identified root package name, e.g., when the original library root package `com.gson` was obfuscated/renamed to `com.mygson` or `com.ab`, we rename the original library API accordingly. For ambiguous profile matches, i.e., *LibScout* is not able to distinguish patch-level changes in libraries, we select one of the matched libraries. Since patch-level changes are API-compatible this does not affect the subsequent updatability check.

3. Library updatability: Finally, we combine these two data sets to determine whether and to which extent libraries in apps can be updated. While libraries can, by definition, be replaced by patch and minor releases, this large-scale analysis investigates whether libraries can be replaced by subsequent major versions that account for 44% of all library releases. Furthermore, this allows us to identify hotspot-APIs, i.e., APIs of the libraries that are most/least frequently used, and to determine their stability.

4.2 Updatability Statistics

We conduct a large-scale evaluation in which we analyzed 98 distinct libraries and scanned 1,264,118 apps from Google Play. The results are summarized in Figure 13. LibScout successfully identifies 2,028,260 libraries (exact matches only). In 239,019 cases (11.8%), we could not detect any library APIs that are actively used, i.e., those libraries are dead code. For the remaining 1,789,241 libraries, we can determine the set of used APIs and correlate it with the API robustness data. The results suggest that in 85.6% of the cases the identified library can be upgraded by at least one version (*Upgrade1+*) without any code adaption, simply by replacing the old library. Even more surprising, a subset of 861,852 libraries (48.2%) can be upgraded to the most current library version (*Upgrade2Max*). Only in 14.4% of the cases the library can not be upgraded by a single version without additional effort (*non-upgradable*), i.e., the next version changed or removed used APIs. One major reason for this high updatability rate is that although the majority of libraries offer hundreds or even thousands of different API functions, the typical app developer only uses a small subset thereof. Our results indicate that the average number of APIs used across libs is 18.

Table 5: Updatability to the most current version by sum of libraries and library matches grouped into 20% bins. How to read: Between 80–99% of all identified versions of 10 distinct libraries can be upgraded to the latest version. These 10 libraries account for 579,294 library matches.

Percentage	by # of libs	by # of lib matches
100%	5 (13.5%)	11,346 (1%)
80–99%	10 (27%)	579,294 (51%)
60–79%	5 (13.5%)	139,189 (12.3%)
40–69%	5 (13.5%)	121,671 (10.7%)
20–39%	4 (10.8%)	228,393 (20.1%)
0–19%	8 (21.6%)	55,690 (4.9%)
Total	37	1,135,583

In the following, we analyzed the extent to which libraries in apps could be upgraded to the *latest* version. In contrast to the SemVer analysis in the previous section, this puts a higher focus on the robustness of more popular APIs. Libraries that are stable in their most frequently used APIs, even across major versions, are assumed to have a high-updatability rate. To verify this assumption, we grouped 37 libraries for which we have more than 10 versions and more than 50 matches in our large-scale analysis according to updatability to the newest version. Table 5 shows the fraction of library matches that can be updated to the latest version, bucketed into 20% bins. The column on the right aggregates the absolute numbers of matches for those libraries.

We could not find a correlation between the absolute number of used APIs and library updatability. While the libraries in the top bucket on average use 11.9 APIs (with a standard deviation of $\sigma = 7.8$), the libraries in the last bucket only have a slightly higher API usage (mean = 14.7, $\sigma = 8.4$). However, aggregating the top ten most frequently used library functions and correlating them with their stability across library versions revealed the root cause. Libraries with a high updatability to the most current version are stable for the most popular APIs (even across major versions), while libraries with a very low updatability showed a completely

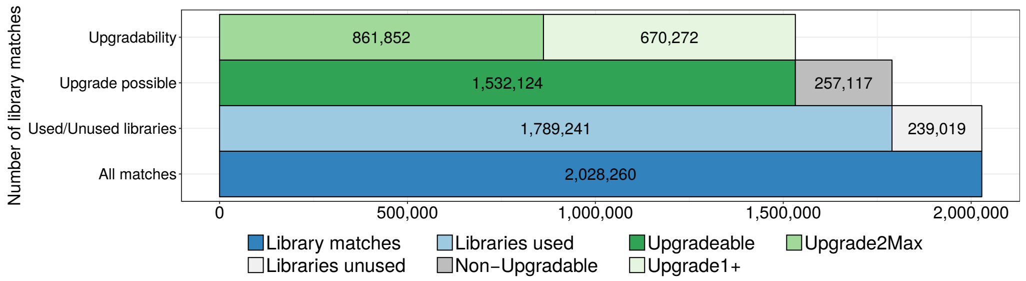

Figure 13: Library updatability of current apps on Google Play

different picture. In these cases, either (parts of) the most popular APIs have been completely replaced by a new APIs or existing ones have been modified or renamed. In our data set, Google's *Gson* library was a positive example with a 99.91% updatability to the most current version in 315,079/315,371 library matches. On the other hand, *Retrofit* could have been upgraded to the latest version only in 0.07% (20/30,568) cases due to major API changes in recent versions.

4.3 Security Vulnerability Fixing

Besides general library updatability, we are particularly interested in how easy vulnerable library versions can be patched. To this end, we investigate the eight publicly known vulnerabilities described in Section 3. We could not easily increase the set of libraries since it is not trivial to find reports on SDK vulnerabilities (see Table 4). However, the affected libraries are commonly used by many applications (14.4% of 1,264,118) and thus any vulnerability in these libraries does affect thousands or even millions of users.

Table 6 shows the libraries with the range of vulnerable versions. After scanning the app repository, we found 18,397 apps that include one of the vulnerable library versions. This is particularly surprising for the vulnerabilities reported by ASI, since the remediation deadline for those libraries has already expired[4], i.e., many app developers either have not reacted to Google's reporting or did not receive a notification in the first place. The subsequent API usage analysis revealed that 91.5% of these libraries are actively used by applications, i.e., at least one API call to the library was found in the non-library code. In the remaining 8.5% of cases the library is included in the app but is not in use, i.e., it is considered dead code. This number is slightly lower than the 11.8% reported for all libraries. For the advertising library *MoPub* we were not able to find apps with one of the vulnerable library versions from the year 2015. Note that using some of these libraries is already sufficient to be vulnerable. This includes all advertisement libraries and Dropbox. There is no need to explicitly invoke specific APIs, since the library's core functionality (*Dropbox* authentication or showing ads in a `WebView`) is triggered upon initialization without further interaction. For the remaining libraries the vulnerable functionality has to be triggered by the application such as login at *Facebook* or certificate pinning from *OkHttp*.

[4]Apps are not deleted from Google Play after the remediation phase but further app updates are rejected as long as the vulnerability remains unfixed.

Out of the 16,837 actively used libraries, 97.8% could be patched through a simple drop-in replacement of the vulnerable version with the fixed one. In 57.3% of the cases, the library could even be replaced by the most current version available. The perfect updatability result for *Airpush* is due to the fact, that the patched version is the most current version to date and includes code changes only. Therefore, all versions of the second-to-last version 8.0 could be upgraded to the latest version. *Dropbox* achieved the lowest auto-fix rate since there were some changes to the most frequently used APIs between 1.5.4 and the fixed version 1.6.2. Note, that the actual numbers for *Airpush* and *Vungle* could even be higher, since we were only able to retrieve between 1–3 versions prior to the fixed version.

5 DISCUSSION

In the Android app ecosystem the majority of developers makes an increasing use of third-party libraries to enhance usability and functionality of their apps. However, those components are a double-edged sword. While alleviating development through code reuse, they have been found to be a major source of bugs and security vulnerabilities [17, 38, 40, 42]. To provide end-users reliable software, it is therefore of outmost importance to keep third-party libraries up-to-date. However, recent studies [4, 9, 23, 31] have demonstrated that in reality, we are far from having up-to-date third-party components and as a consequence this ultimately puts the end-user's privacy and security at risk.

Our app developer survey (see Section 2) was a first step towards identifying the root causes *why* developers do not update libraries. A valuable insight is that while about 60% of app developers regularly update their application (at least once per quarter), mainly for new functionality, the motivation to update the included libraries is quite low (only 33% considers updating libraries as part of the app update). Contrary to the motivation to update apps for new functionality, the main incentive to update libraries is primarily bugfixes and security fixes. However, this is impeded by the fact that 63% of all library releases mix code fixes with new content and/or non-compatible API changes (cf. Figure 12).

5.1 The Role of the Library Developer

Our survey suggests that many developers abstain from updating dependent libraries due to an expected high integration effort and to prevent incompatibilities. Our library API analysis in Section 3 supports this assumption. There is consistently a mismatch between

Table 6: Number of apps found with a vulnerable library version, number of apps that actively use this library, number of apps that could be patched to the first non-vulnerable version without code adaptation (update2Fix), to the most current version available (update2Max), or not updated to a fixed version without code modification (non-fixable). Unused libraries are not considered in the last three columns.

Library	Vuln. Versions	Matches	Libs in use	update2Fix	update2Max	non-fixable
Airpush	8.0	4,746	4,545	4,545 (100%)	4,545 (100%)	0
Apache CC	3.2.1 / 4.0.0	1,199	749	749 (100%)	502 (67%)	0
Dropbox	1.5.4 - 1.6.1	710	682	410 (60.1%)	6 (0.01%)	272 (39.9%)
Facebook	3.15	1,839	1,808	1,792 (99.1%)	4 (0.22%)	16 (0.88%)
OkHttp	2.1.0 - 2.7.4	7,319	7,179	7,169 (99.9%)	3,013 (42%)	10 (0.14%)
	3.0.0 - 3.1.2	500	237	237 (100%)	236 (99.6%)	0
MoPub	3.10 - 4.3	—	—	—	—	—
Supersonic	5.14 - 6.3.4	1,198	905	905 (100%)	743 (82.1%)	0
Vungle	3.0.6 - 3.2.2	886	732	653 (89.2%)	594 (81.1%)	79 (10.8%)
Total		18,397	16,837	16,460 (97.8%)	9,643 (57.3%)	377 (2.2%)

expected changes, i.e., conveyed through the version number, and the actual changes based on code/API changes (the semantic version was correct only in 42% of all cases). One problem is that some of the library developers are too conservative in that they never increase the major number, e.g., *Digits* (29 versions), *FasterXML-Jackson-Core* (61 versions), or *vkontakte* (29 versions), making the three number versioning scheme an effective two number versioning scheme. Another problem is when different libraries from the same developer, e.g. Android support libraries or Google Play Service libraries, have the same release cycle and different libraries receive the same new version number independent from the actual changes. The main reason for the mismatch between expected and actual semantic version, however, is probably the wrong assessment of changes by the lib developer. This means, that the specification of the patch, minor, or major version is determined by the amount of code changes and effort spent for this update rather than whether the new release is API compatible to the current version.

Another aspect is that 44% of all library updates comprise major versions. This implies that many library developers too frequently release versions that might potentially break application code. This is also backed by survey responses highlighting library update problems like *"It often impacts the rest of the code. Backwards compatibility isn't ensured and that leads to a big effort in updating the libs."* or *"Sometimes library updates break existing features, due to methods changes"*. A more careful API design and aggregating unforced changes like API renaming to fewer major versions would remedy this situation. As highlighted in Section 4, keeping the most frequently used APIs stable, even across major versions, also has a considerable effect on the overall updatability. In particular, library developers should spend more effort in providing dedicated releases for critical bugfixes and security fixes. In six out of ten cases (cf. Table 4), security fixes were even bundled as major release, which severely impedes widespread adoption.

As there is no widely accepted library market place or package manager for Android, changelogs are typically the main means to communicate changes to the application developer. Since about 80% of app developers read changelogs at least from time to time, this is a good way to provide detailed information on bugfixes and API changes. However, in reality, the majority of changelog entries advertises new functionality rather than reporting (detailed) bug fixes. The fact that we could only find a single entry *security fix* illustrates the current status quo pretty well. It seems that library developers put their main focus on functionality that, according to our survey results, is only the third most important update criterion. A recent study [26] reports that API changes/removals in the Android SDK typically trigger discussions on Stack Overflow. A simple means of providing community support after major releases would involve an active participation of library developers in such discussions to clarify changes and provide guidelines on how to perform the upgrade.

There has also been some discussion about the usefulness of Semantic Versioning. While it is certainly not supposed to be *the* gold standard, it is, in fact, a simple and useful means for library developers to express compatibility and for consumers to quickly assess the expected library integration effort. Almost all libraries in our dataset already use the `X.Y.Z` scheme, however, it seems that API compatibility is not always the main factor in the versioning process. An open question remains how many developers are aware of concepts like SemVer and would be able to interpret version changes correctly. To raise the awareness, library developers could pro-actively promote SemVer compliance, e.g. by adding a *SemVer compliant* badge to their code repository. In the long term, this concept will likely become more known, at least among iOS developers, since the new *Swift* package manager enforces versioning according to SemVer rules.

5.2 How to improve Library Updatability?

Based on the survey responses and the follow-up analyses there are different possibilities on how to improve library updatability for different entities of the app ecosystem. Note, that this section is giving educated advices and actionable items based on first-hand information of app developers and results aggregated from follow-up analyses on libraries and apps from Google Play. Implementation, evaluation and assessment of developer adaption for the proposed technical solutions are subject to future work.

The marketplace. One possibility to improve adaption is a centralized marketplace, like the Google Play Store. With the App Security Improvement program, Google introduced a service that identifies security problems in apps. It notifies the respective app developer and provides a support document on how to fix the problems. However, this service also enforces that security fixes are deployed within reasonable time. While this helps to improve the overall application security on the market, it also comes with inherent limitations. It only warns about known vulnerabilities and app developers that are writing apps for markets other than the Play Store do not benefit. The main limitation is, however, that it only fights the symptoms and does not tackle the underlying problem of the poor library version adaption rate.

About 79% of the developers in the survey could think of a dedicated library store or package manager for Android. There are already established package managers for other ecosystems such as *nuget* (.net), *npm* (JavaScript), *Cargo* (Rust), or *Cocoapods* (iOS). There is no equivalent in size and acceptance for Android to search for libraries to date. This is also documented by our survey in which the majority of developers simply refers to *"Google"* or *"Internet"* when being asked where to search for libraries. An accepted central solution could also enforce certain library requirements or quality standards more easily. For instance, the new *Swift* package manager [2] (for macOS and soon for iOS) expects packages to be distributed as source and to be named according to SemVer rules. Source distributions might foster contributions and creation of patches through the community. This is also backed by the results of the survey in which open-source is the main criteria for library selection for 61% of developers, next to functionality with about 80%.

Development tools. In 2014 the Android Gradle plugin was introduced to give app developers a powerful dependency manager to facilitate building complex applications with a larger number of third-party components. But although this is the preferred way to integrate libraries for about 30% of app developers, there is still a high number (20%) that manually integrates libraries or uses a combination of different methods (33%). Despite Gradle's high acceptance (64% like its usability, 31% somewhat), the main criticism constitutes its poor performance and the steep learning curve that might be reasons to resort to different approaches. Google picked up this criticism and recently announced a new Gradle version for Android Studio that particularly improves build times for complex applications [6]. Another argument against mixed approaches is that including libraries manually implies a higher update effort, since new versions have to be downloaded manually and there is no notification when new releases become available. This reinforces the unawareness of library updates among app developers as shown in Figure 9. In contrast, Android Studio 2.2 recently integrated an opt-in feature to automatically notify app developers when updates of integrated third-party libraries from remote repositories such as *Maven Central* and *JCenter* become available.

Besides improving the dependency manager, integrating our detection of library version compatibility into the IDE could be helpful to automatically classify a library update and to inform about the expected code adaption effort based on the set of used library APIs. We are currently in the process of evaluating how such a plugin could be implemented for Android Studio, the preferred IDE for about 61% of app developers in our survey.

Automated library updates to the rescue? A prominent example for auto-updates is the former system component `WebView` that was moved to a standalone-app in Android 5.0 after a series of severe security vulnerabilities. Distributed as an app, Google can automatically push security patches to this commonly used component to reach millions of devices that do no longer receive Android OS updates. Similarly, the app update mechanism of Google Play was adapted to install app updates automatically as long as no new permissions are requested. However, patching libraries that are part of the application bytecode is somewhat more challenging. Although Section 4 has demonstrated that 85.6% of libraries could be automatically updated, in 48.2% of the cases even to the latest version, there might be additional obstacles that prevent auto-updates of minor and major releases in reality (cf. Section 5.3). However, limiting auto-updates to patch versions that provide critical bugfixes and security fixes, would already tremendously improve the current status-quo.

One possible integration approach includes the developer specifying a subset of included libraries eligible for automatic updates. A similar approach is deployed by Google Chrome to automatically update extensions [18]. The difference, however, is that the extension developer may specify this flag. There is also no formal requirement or quality assurance required, since, in worst case, the extension could simply be disabled after an unstable update. In Android, one could further introduce an option to limit updates to patch level updates that do not introduce new functionality. According to the survey, 52% of app developers would welcome such an automated update mechanism, while only one quarter disapproves such approaches. Note, that the questionnaire asked for updates in general, not for bugfix/security fix updates in particular. Thus, the acceptance for auto-fixing critical bugs only might actually be higher.

There are also different on-device deployment strategies to integrate new library versions. One option that does not require larger modifications of the app installation routine is to integrate new libraries during on-device compilation time. In Android 6, the ahead-of-time compiler on the device compiles the entire applications' bytecode to native code. There, the compilation would have to be re-triggered for each library update, similar as for new app updates. With Android 7, the ahead-of-time compilation was replaced by a just-in-time compiler [13]. This way, library code could be updated through a forced re-compilation whenever such code is used by the app. Given the generally low library API usage, the expected compilation overhead should be negligible.

Decoupling library code from application code, i.e., moving to dynamic linking, would be another option to facilitate library updates. Dynamic linking of third-party components has been disallowed by Android and iOS for a long time due to security reasons. This changed with iOS 8, released in September 2014, when Apple addressed these security concerns with a new kernel extension to check the integrity of app files [1], i.e., whether dynamic libraries are signed, have a valid Team Identifier and that this identifier matches the one of the containing application. Updating (compatible) libraries is then simplified to replacing the library file.

5.3 Threats to Validity

Programmatically determining the public API of a software component is a non-trivial task, specifically on Android where, among others, advertisement libraries are typically obfuscated with identifier renaming. We distinguish obfuscated and non-obfuscated names in a best effort approach, but for corner cases this is generally undecidable. The same is true for whitelisting package names that are supposed to be used internally. Only the ground truth in form of a proper documentation for all library releases would provide the complete public interface. However, this information is not always available. Given that we are conservative in our filtering list, we report a lower bound on updatability, e.g., when we erroneously include an obfuscated public API which is not present in the successor version due to re-obfuscation.

We conduct our library updatability analysis based on API compatibility. This constitutes the main factor to determine whether a library can be updated without any code adaption. We do not include rare cases in which public, static class fields are renamed in subsequent library releases. While such cases can cause incompatibility, we assume that they do not occur frequently. Moreover, we do not assess whether the intended functionality is preserved in the new release, i.e., that no new bugs are introduced and no code semantics changed that cause unexpected side-effects. Changing semantics of already existing APIs is considered bad practice and strongly discouraged as there is no simple means of detecting such cases for the library consumer.

We also consider the case when libraries depend on additional libraries. Versions that include other libraries can only be updated if all sub-dependencies can be updated as well. We found that 55% of the libraries in our database include at least one version with sub-dependencies. However, through manual investigation, we identified most of these dependencies as optional. In the majority of cases, advertisement mediation frameworks can be configured to use multiple ad libraries from different providers. There are two utility libraries that are used by five other libraries, *Gson* and *okio* with an updatability of 99.9% and 100%, respectively. Hence, it is safe to assume that these sub-dependencies do not influence the updatability of libraries that include them.

Finally, we investigated changes of the minimal, required Android API level by libraries. Although this does not affect the correctness of our library updatability results, the app developer might have to increase the app's minimum API level in order to update a library. This implies that the updated application may no longer be compatible with devices having an older Android version which consequently reduces the app's potential user base. To investigate the severity of such cases we aggregated a history of changes of the minimal API for the eight libraries in Table 6 from publicly available changelogs. Across versions, libraries have changed the minSDK version between 1–2 times. The most current version of five libraries requires a minimum API level between 11–16 (Android 3.0–4.1). *Dropbox* does not state this requirement explicitly, only indirectly via an Android sample (API level 19, Android 4.4). According to the latest Google Play Access Statistics [15] less than 2% of all users have a device with API level < 16 (9% with API level < 19). These results suggest that library developers are very conservative in their choice of the minimal SDK to support a wide range of devices and consequently the expected loss of potential users is negligible for app developers.

6 RELATED WORK

There have been several studies on different software ecosystems to assess the ripple effect of API changes. Dig et al. [16] found that in 80% of cases API changes in libraries break the client application upon update. Kim et al. [22] investigated the relationship between library API changes and bugs. They found that the number of bugs particularly increases after API refactoring. Bavota et al. [5] studied the evolution of dependencies between Java projects of the Apache ecosystem to find that client projects are more willing to upgrade a library when the new version includes a high number of bugfixes. At the same time, API changes discouraged the user from upgrading since substantial code adaption effort might be required to include the new release. While those findings are in line with our results, i.e., mismatch of expected and actual changes and insights from app developers about why libraries are not updated, this work goes one step further. We identified root causes for this problem in the Android ecosystem. Based on our results we thoroughly discussed various options to remedy this situation that would have a high app developer acceptance (based on our survey results).

McDonnell et al. [29] studied the Android API stability and adoption and found that app developers do not quickly adopt new APIs to avoid instability and integration effort. Another study on the Android API [25] showed that including fast-changing and error-prone APIs negatively affects the app ratings in the market. In contrast to our work, these studies investigated the Android API, however, we can confirm their findings for third-party libraries as well. Particularly, for over-privileged libraries, app developers often receive negative feedback and ratings, e.g. *"some users have complained about the permissions the app requires due to libraries"* or *"Google Play services and especially maps required for sometime the storage permission which led to lots of questions and negative ratings"*.

Various studies [21, 30, 37] emphasized that code reuse is widespread in the Android market and that third-party libraries account for most of it [24, 27]. At the same time, the number of critical bugs and security vulnerabilities in third-party components has steadily increased. Over the last years they have become the weakest link and the prime attack vector of applications [38]. Dedicated research [8, 19, 35, 39] has particularly found advertisement libraries to be hazards for the end-users' security and privacy by secretly collecting private data or even opening backdoors. This has motivated a line of research to automatically detect libraries in applications [10, 28, 32, 41]. However, these early approaches were insensitive to exact library versions. More recent studies [4, 9] adopted Software Bertillonage techniques [11] to identify concrete library versions and to uncover that about 70% of included libs in Google Play apps are outdated by at least one version. Similar alarming results have recently been reported for other ecosystems, such as Javascript [23] and Windows [31].

As a consequence, fast response times by library developers remain ineffective and even known security vulnerabilities [3, 7, 33–35] remain a persistent threat in the app ecosystem, when app developers take on average more than 300 days to integrate the existing fixes [4]. While previous work focused on detecting outdated libraries, this work seeks to find the root causes. Ultimately, our findings allowed us to propose actionable items that are both effective in amending the library outdatedness problem and accepted by the majority of app developers.

7 CONCLUSION

With the rapidly increasing number of used libraries, large parts of Android apps consist of third-party code. Critical bugs and security vulnerabilities in such components reach a high number of end-users and put their privacy and sensitive data at risk. At the same time reality shows that important patches either reach the app consumer only after an unacceptable long period or not at all. This paper is the first to identify the root causes of *why* Android app developers do not adopt new versions. Based on first-hand information of app developers and results of two empirical studies, we propose actionable items for different entities of the app ecosystem to remedy this alarming situation. We belief that tackling the underlying problem is more effective than fighting the symptoms. This approach is also preferred by Derek Weeks (vice president at Sonatype) when being asked for a long-term solution: *"It's not a story about security professionals solving the problem, it's about how we empower development with the right information about the (software) parts they are consuming."*

Ethical considerations. We reported apps that include a vulnerable library version to Google's ASI program.

ACKNOWLEDGMENTS

This work was supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) (FKZ: 16KIS0345, 16KIS0656) and the project SmartPriv (FKZ: 16KIS0377K).

REFERENCES

[1] Apperian. 2014. The Impact of iOS 8 on App Wrapping. https://www.apperian.com/mam-blog/impact-ios-8-app-wrapping. (2014). Last visited: 08/25/2017.

[2] Apple. 2016. Swift Package Manager Community Proposal. https://github.com/apple/swift-package-manager/blob/master/Documentation/PackageManagerCommunityProposal.md. (2016). Last visited: 08/25/2017.

[3] Google ASI. 2016. Security Vulnerability in Vungle Android SDKs prior to 3.3.0. https://support.google.com/faqs/answer/6313713. (2016). Last visited: 08/25/2017.

[4] Michael Backes, Sven Bugiel, and Erik Derr. 2016. 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)*. ACM, New York, NY, USA, 356–367.

[5] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache Community Upgrades Dependencies: An Evolutionary Study. *Empirical Softw. Engg.* 20, 5 (Oct. 2015), 1275–1317.

[6] Android Developers Blog. 2017. Android Studio 3.0 Canary 1. https://android-developers.googleblog.com/2017/05/android-studio-3-0-canary1.html. (2017). Last visited: 08/25/2017.

[7] Dropbox Blog. 2015. Security bug resolved in the Dropbox SDKs for Android. https://blogs.dropbox.com/developers/2015/03/security-bug-resolved-in-the-dropbox-sdks-for-android. (2015). Last visited: 08/25/2017.

[8] Theodore Book, Adam Pridgen, and Dan S. Wallach. 2013. Longitudinal Analysis of Android Ad Library Permissions. In *MoST'13*. IEEE.

[9] Zhihao Mike Chi. 2016. LibDetector: Version Identification of Libraries in Android Applications. (August 2016).

[10] Jonathan Crussell, Clint Gibler, and Hao Chen. 2013. Andarwin: Scalable detection of semantically similar android applications. In *ESORICS'13*. Springer.

[11] Julius Davies, Daniel M. German, Michael W. Godfrey, and Abram Hindle. 2011. Software Bertillonage: Finding the Provenance of an Entity. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*. ACM, New York, NY, USA, 183–192.

[12] Android Developers. 2015. App Security Improvement Program. https://developer.android.com/google/play/asi.html. (2015). Last visited: 08/25/2017.

[13] Android Developers. 2016. Android 7 for Developers. https://developer.android.com/about/versions/nougat/android-7.0.html. (2016). Last visited: 08/25/2017.

[14] Android Developers. 2017. App Security Improvements: Looking back at 2016. https://android-developers.googleblog.com/2017/01/app-security-improvements-looking-back.html. (2017). Last visited: 08/25/2017.

[15] Android Developers. 2017. Google Play Dashboard. https://developer.android.com/about/dashboards/index.html. (2017). Last visited: 08/25/2017.

[16] Danny Dig and Ralph Johnson. 2006. How Do APIs Evolve? A Story of Refactoring: Research Articles. *J. Softw. Maint. Evol.* 18, 2 (March 2006), 83–107.

[17] Hewlett Packard Enterprise. 2016. HPE Cyber Risk Report. https://techbeacon.com/resources/2016-cyber-risk-report-hpe-security. (2016). Last visited: 08/25/2017.

[18] Google. Last visited: 02/10/2017. Chrome Extensions Autoupdating. https://developer.chrome.com/extensions/autoupdate. (Last visited: 02/10/2017).

[19] Michael Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *WISEC'12*. ACM.

[20] GuardSquare. 2016. ProGuard Java Obfuscator. http://proguard.sourceforge.net. (2016).

[21] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. 2013. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *DIMVA'12*. Springer.

[22] Miryung Kim, Dongxiang Cai, and Sunghun Kim. 2011. An Empirical Investigation into the Role of API-level Refactorings During Software Evolution. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 151–160.

[23] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '17)*.

[24] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. An investigation into the use of common libraries in android apps. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*, Vol. 1. IEEE, 403–414.

[25] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. API change and fault proneness: a threat to the success of Android apps. In *Proceedings of the 9th joint meeting on foundations of software engineering (ESEC/FSE '13)*. ACM, 477–487.

[26] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2014. How Do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014)*. ACM, New York, NY, USA, 83–94.

[27] Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2014. Revisiting Android Reuse Studies in the Context of Code Obfuscation and Library Usages. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 242–251.

[28] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps. In *ICSE'16*. ACM.

[29] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. IEEE Computer Society, Washington, DC, USA, 70–79.

[30] Israel J Mojica, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E Hassan. 2014. A large-scale empirical study on software reuse in mobile apps. *IEEE software* 31, 2 (2014), 78–86.

[31] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. 2015. The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching. In *Proc. 36th IEEE Symposium on Security and Privacy (SP '15)*. IEEE, 692–708.

[32] Arun Narayanan, Lihui Chen, and Chee Keong Chan. 2014. Addetect: Automated detection of android ad libraries using semantic analysis. In *ISSNIP'14*. IEEE.

[33] The Hacker News. 2014. Facebook SDK Vulnerability Puts Millions of Smartphone Users' Accounts at Risk. http://thehackernews.com/2014/07/facebook-sdk-vulnerability-puts.html. (2014). Last visited: 08/25/2017.

[34] The Hacker News. 2015. Backdoor in Baidu Android SDK Puts 100 Million Devices at Risk. http://thehackernews.com/2015/11/android-malware-backdoor.html. (2015). Last visited: 08/25/2017.

[35] The Hacker News. 2015. Warning: 18,000 Android Apps Contains Code that Spy on Your Text Messages. http://thehackernews.com/2015/10/android-apps-steal-sms.html. (2015). Last visited: 08/25/2017.

[36] Tom Preston-Werner. 2013. Semantic Versioning 2.0.0. http://semver.org/. (2013). Last visited: 08/25/2017.

[37] Israel J Mojica Ruiz, Meiyappan Nagappan, Bram Adams, and Ahmed E Hassan. 2012. Understanding reuse in the android market. In *Proceedings of the 20th*

International Conference on Program Comprehension (ICPC '12). IEEE, 113–122.

[38] Sonatype. 2017. 2016 State of the Software Supply Chain. https://www.sonatype.com/software-supply-chain. (2017). Last visited: 08/25/2017.

[39] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. 2012. Investigating User Privacy in Android Ad Libraries. In *MoST'12*. IEEE.

[40] ThreatPost. 2016. Code reuse - A peril for secure software development. https://threatpost.com/code-reuse-a-peril-for-secure-software-development/122476/. (2016). Last visited: 08/25/2017.

[41] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. WuKong: A Scalable and Accurate Two-phase Approach to Android App Clone Detection. In *ISSTA'15*. ACM.

[42] Jeff Williams and Arshan Dabirsiaghi. 2012. The unfortunate reality of insecure libraries. http://www.aspectsecurity.com/research-presentations/the-unfortunate-reality-of-insecure-libraries. (2012). Last visited: 08/25/2017.

A APPENDIX: QUESTIONAIRE

Professional Background Questions:
B1: Is developing Android apps your primary job?
(i) yes, (ii) no
B2: Are you developing your apps as a hobby, are you self-employed or do you work for a company? Please check all that apply.
(i) hobby, (ii) self-employed, (iii) company, (iv) other
B3: How large is your company?
(i) up to 10 employees, (ii) 10-50 employees, (iii) 50-100 employees, (iv) >100 employees
B4: How many apps have you worked on?

App Development Questions:
A1: How do you develop your app/apps? (If more than one, please choose the one you use primarily)
(i) Android Studio, (ii) Eclipse, (iii) Application Generator Framework (Cordova, Xamarin,...), (iv) other
A2: Is/Are your app/apps updated on a fixed schedule?
(i) yes, (ii) no
A3: Which intervals do you use to update your app/apps?
(i) weekly, (ii) bi-weekly, (iii) monthly, (iv) quarterly, (v) twice per year, (vi) yearly, (vii) never
A4: For which purpose do you update your app/apps? Please check all that apply.
(i) new functionality, (ii) bugfixes, (iii) library updates, (iv) other

Third-Party Library Questions:
T1: Where do you search for the libraries?

T2: Do you choose libraries according to specific criteria? Please check all that apply.
(i) Popularity, (ii) Functionality, (iii) Open-Source, (iv) Closed-Source, (v) Required Permissions, (vi) Documentation, (vii) Recommendations, (viii) Ratings, (ix) Security, (x) Update frequency, (xi) other
T3: How many different library functions do your apps typically use?

T4: How do you integrate third-party libraries into your app? Please check all that apply.
(i) Add JAR file, (ii) Gradle, (iii) Ant, (iv) Maven, (v) I don't know, (vi) other
T5: Are you happy with gradle's usability?
(i) yes, (ii) somewhat, (iii) no, (iv) I don't know
T6: Can you list a few problems that you've had with gradle?
T7: Do you update the libraries in your app regularly?
(i) yes, all of them, (ii) yes, some of them, (iii) no, (iv) I don't know
T8: Why do you update your apps' libraries?
(i) New features, (ii) Bugfixes, (iii) Security fixes, (iv) I don't know, (v) other
T9: If your app were to contain outdated libraries, why would that be? Please check all that apply.
(i) Library was still working, (ii) Too much effort, (iii) Missing update documentation, (iv) Unaware of updates, (v) Prevent incompatibilities, (vi) Bad/missing library documentation, (vii) I don't care, (viii) I don't know, (ix) other
T10: Do you have positive/negative examples for libraries regarding updatability, documentation etc.? Please give details.

T11: Would you welcome automatic library updates on user devices via the Android OS in cases where they do not break functionality?
(i) yes, (ii) no, (iii) I don't mind, (iv) I don't know
T12: Which of the following do you think would help make library updates easier? Please check all that apply.
(i) Different distribution channels, (ii) Central library marketplace, (iii) Better IDE integration, (iv) System service or package manager, (v) other
T13: Have you ever encountered negative feedback/ratings solely because of included library functionality (e.g. libs that perform tracking or aggregate user data)?
(i) yes, (ii) no, (iii) I don't know
T14: What was the problem?

Demographics:
D1: How old are you? Enter 0 if you don't want to answer

D2: What is your gender?
(i) male, (ii) female, (iii) I don't want to answer
D3: What is your highest educational degree?
(i) High school, (ii) College degree, (iii) Graduate degree, (iv) I don't want to answer, (v) No degree
D4: How many years of general coding experience do you have?

D5: How many years of Android experience do you have?

D6: How did you learn to write Android code? Please check all that apply.
(i) Self-taught, (ii) Class in school, (iii) Class in university, (iv) On the job, (v) Online coding course, (vi) Other