Up2Dep: Android Tool Support to Fix Insecure Code Dependencies

Duc Cuong Nguyen, Erik Derr, Michael Backes, and Sven Bugiel
CISPA Helmholtz Center for Information Security
{duc.nguyen,erik.derr,backes,bugiel}@cispa.saarland

ABSTRACT
Third-party libraries, especially outdated versions, can introduce and multiply security & privacy related issues to Android applications. While prior work has shown the need for tool support for developers to avoid libraries with security problems, no such a solution has yet been brought forward to Android. It is unclear how such a solution would work and which challenges need to be solved in realizing it.

In this work, we want to make a step forward in this direction. We propose Up2Dep, an Android Studio extension that supports Android developers in keeping project dependencies up-to-date and in avoiding insecure libraries. To evaluate the technical feasibility of Up2Dep, we publicly released Up2Dep and tested it with Android developers (N=56) in their daily tasks. Up2Dep has delivered quick-fixes that mitigate 108 outdated dependencies and 8 outdated dependencies with security problems in 34 real projects. It was perceived by those developers as being helpful. Our results also highlight technical challenges in realizing such support, for which we provide solutions and new insights.

Our results emphasize the urgent need for designated tool support to detect and update insecure outdated third-party libraries in Android apps. We believe that Up2Dep has provided a valuable step forward to improving the security of the Android ecosystem and encouraging results for tool support with a tangible impact as app developers have an easy means to fix their outdated and insecure dependencies.

KEYWORDS

1 INTRODUCTION
Software developers commonly re-use existing code, in particular in the form of third-party libraries. Third-party libraries are software components that are bundled in a form that can be distributed to developers through different channels, such as central repositories. However, those libraries come at a cost: if they contain bugs or security and privacy issues, those flaws could be amplified by being integrated in different applications that use the affected library versions. Prior works [16, 19, 22, 33, 38] showed that such libraries could expose user sensitive information to third-party applications, or be a contributing factor for cryptographic API misuse in applications. More concerning, even when privacy & security related fixes were available in newer versions of affected libraries, their adoption by developers progressed very slowly [14]. Existing work has proposed different solutions to overcome the problem of outdated third-party libraries. Ogawa et al. [40] proposed using an external service to split app code and third-party library code from Android application package (APK) files, and then replace the (vulnerable) outdated libraries with their fixed versions. This might improve the situation but requires user actions to re-install the updated APK. Market stores may play the central role to roll out updates for libraries to end users but third-party libraries are tightly integrated into their host app which makes it virtually impossible to precisely separate app code and library code [14], and to pinpoint the exact version of a library that an app is using. App developers (not market stores, not end users) are in the perfect position to fix this problem in today’s ecosystem as app code and library code are separated in their development environment. They are aware of the exact library version they are using and can upgrade their dependencies. However, developers do not update their app’s dependencies [19] because the outdated libraries are still working; developers fear incompatibilities between library versions, specifically semantic versioning has been found unreliable and has failed developers; developers are unaware of the updates; and updates take too much effort. This raises an important issue: providing security updates does not solve the problems at all if developers do not migrate their app’s dependencies to the security fixes.

Unfortunately, existing solutions to support developers in keeping their project’s dependencies up-to-date are ineffective. Android Studio itself includes Lint tool [3] to inform developers about outdated third-party dependencies in Android projects. However, Lint only provides developers information on whether there are newer versions of the included libraries, but it does not alert developers about security vulnerabilities of the libraries and it is limited to a list of only two libraries that are classified as privacy risks and without further information about the risks. The lack of information on whether the current version is secure and on whether the newer version is compatible with the existing code of the app makes
developers afraid of migrating their project dependencies to newer versions, and, more importantly, it accustoms developers to stay with the current (although outdated and more likely vulnerable) versions as long as the app is still working.

With established tools being unreliable and not universally adopted (semantic versioning) [19] or providing insufficient support (Lint), this leaves a gap between developers’ expressed wishes for support and the status quo. At the same time, there are still open questions on the technical feasibility of tool support for developers to keep project’s dependencies up-to-date, which challenges come along the way, especially how developers would receive (i.e., apply) such a tool support, and most importantly its (security) impact on real projects. As long as these questions are not yet answered, we will still see security & privacy problems in apps that are attributed to outdated, insecure third-party libraries. Therefore, to make advances in filling this gap with appropriate tool support for app developers, we focus in this paper on the following research questions: “Would it be technically feasible to support developers in keeping their project’s dependencies up-to-date?” and, more importantly, “Could such a tool support have a tangible impact on the security and privacy of Android apps?” To try to answer these questions, we developed an Android Studio extension called Up2Dep to help Android app developers in upgrading their library dependencies and in avoiding vulnerable library versions. Up2Dep analyzes third-party libraries to provide developers with information about the changes that they may need to perform when updating a library, based on the public API changes between the library versions. Using the collected information about library APIs and their usages on a given project, Up2Dep provides feedback on the updatability of outdated library versions (i.e., we base our updatability information only on the code itself and not on unreliable external sources like semantic versioning of libraries).

Up2Dep also maintains a database of publicly disclosed vulnerabilities and cryptographic API misuse of libraries, and alerts developers if a vulnerable library version is included in their apps.

Our solution is the first implemented solution to support app developers in their task to avoid outdated, critical dependencies, and an important step to gather first-hand feedback from developers about solutions that so far have only been recommended in the literature.

We tested Up2Dep with developers to see how Up2Dep can support them in their daily programming tasks. To measure the impact of Up2Dep, we implemented telemetric features inside Up2Dep that gather anonymized information on how developers interact with Up2Dep and that allow developers to provide feedback in-situ on whether the suggested quick-fixes worked as they expected and what they think about such support from Up2Dep. Our telemetric data shows that among 56 developers, 30 have applied quick-fixes suggested by Up2Dep on 34 real projects, totaling 116 quick-fixes (8 insecure library versions, 108 outdated library versions). The results from the 60 in-situ feedbacks we received from 22 developers confirm that 80.0% of the proposed fixes worked and Up2Dep’s support was useful, while only four cases of the proposed fixes did not work.

Upon further investigating the feedback, we discovered that 13.51% libraries in our dataset have hidden security related problems as the problems reside in the transitive dependencies of those libraries, and are not shown to the developers. We believe, this is a new and important finding because if this problem is not solved, many app developers would continue using insecure libraries without being aware of it. This is detrimental for the security of the Android ecosystem as end users of such apps will eventually be exposed to a variety of attacks. Therefore, we subsequently developed a solution to tackle this problem by analyzing and alerting developers of libraries that have (hidden) transitive dependencies with security problems. Further, our study results show that having tool support on the compatibility of the updates really helps developers more willing to keep their project’s dependencies up-to-date. Lastly, we further evaluated developers’ Up2Dep experience in an online survey where 23 developers shared with us their opinion. Up2Dep received a SUS score [17] of 76.20, which indicates that Up2Dep was considered good by developers in terms of usability.

In summary, we make the following tangible contributions:

- We significantly extended LibScout’s original library dataset (by the factor of 7.5x, totaling 1,878 libraries with their complete version history) and analyze those libraries (37,402 library versions) to discover cryptographic API misuse. To support future research, we make both Up2Dep and this dataset publicly available.

- We built an Android Studio extension called Up2Dep to warn developers about vulnerable library versions including both publicly disclosed vulnerabilities and cryptographic API misuse. Up2Dep helps developers upgrade their project’s dependencies, taking into account the library API compatibility.

- We evaluated the technical feasibility of Up2Dep with Android developers (N=56) in-the-wild and gather anonymized usage information with our telemetric features. Our results show that Up2Dep has helped developers in fixing their project dependencies (n=108) and in avoiding dependencies with security problems (n=8). The majority (80.0%) of suggested fixes (from developer’s feedback) worked and developers found them useful, while only four instances of the proposed fixes did not work as developers expected.

- We discovered that 13.51% of the libraries (233 out of 1,725) have hidden security problems by including (insecure) dependencies which is normally not visible to developers. We have subsequently developed a solution to tackle this problem.

- Our results show that developers indeed are in favor of such support and are willing to use it in their projects. Thus, this work makes a call for action to include such an IDE-provided support for app developers to avoid insecure code dependencies already during app development and for the research community to further investigate how library updatability can be further improved (e.g., detecting non-code, breaking changes between library versions).

2 RELATED WORK

We discuss related works on studying the security of third-party libraries and on tool support for developers in creating more secure apps.

Security of third-party libraries: Sonatype reported that almost 2 billion software components were downloaded per year that contain at least one security vulnerability, and that outdated

1https://github.com/ngcuongst/up2dep
software components had a three times higher rate of security issues [1]. In the web world, Lauinger et al. [32] showed that 37% of 133k analyzed websites include at least one library with a known vulnerability, and that it takes years for web developers to upgrade the included dependencies to the latest version. On the other hand, regarding Android applications, Stevens et al. [44] investigated the user privacy in Android advertisement libraries and found that among 13 investigated ad libraries, several of them are over-privileged. Looking further into Android apps, Backes et al. [14] proposed the LibScout tool to detect third-party library code in Android apps, and found that 70.4% of the included libraries in their dataset are outdated. They also found that it took developers on average almost one year to migrate the app to the latest library version.

Muslukhov et al. [33] proposed BinSight, a static program analyzer that was capable of identifying source attribution in Android applications. The authors revealed that for 90% of the apps that contain cryptographic API misuses, at least one violation originated from third party inclusions. Watanabe et al. [46] also found that 70% and 50% of vulnerabilities of free and paid apps, respectively, stemmed from software libraries, particularly from third-party libraries.

**Tool support for software developers:** Prior work has proposed different approaches and tools to support developers in building more secure Android apps. Among them, many developed tools to find vulnerabilities in applications after they have been released [21, 35, 41]. This means that developers were only aware of such security mistakes at the end of or after their development cycle. Other tools [29, 39] provided developers support while they were writing code. Krüger et al. [29] developed Cognicrypt to support developers in securely using crypto APIs. Rahaman et al. [42] proposed a set of analysis algorithms and a static analysis tool namely Crypto-Guard for detecting cryptographic and SSL/TLS API misuses to help developers analyze large Java projects. Specifically targeting Android applications, Nguyen et al. [39] proposed FixDroid to provide developers with feedback regarding common vulnerabilities while developers write code. Focusing on supporting Android developers in writing more privacy-friendly apps, Li et al. [34] proposed Coconut, an Android Studio plugin that engages developers to think about privacy during app development and to provide real-time feedback on potential privacy issues. Further, Android Studio, Google’s official IDE to develop Android apps, includes Lint tool [3] to check for outdated third-party libraries. Lint, however, only informs developers about whether or not a newer version of the library is available.

None of the above solutions supports developers in keeping their project dependencies up-to-date while taking into account the effort to update the dependencies, the compatibility of the update, or the potential security vulnerabilities of the different dependencies’ versions. While being the Google-provided tool for Android developers, also Android Studio has not considered all these aspects to help developers in keeping their project dependencies up-to-date, and especially to avoid insecure library versions.

![Figure 1: Up2Dep’s architecture. Gray boxes are external components](image)

### 3 UP2DEP DESIGN

In this paper, we propose Up2Dep, an Android Studio extension that facilitates the task of keeping Android project dependencies up-to-date, and help developers avoid insecure library versions. We focus on the Android Studio IDE as this is the tool officially supported by Google and a previous survey [19] has shown that most Android developers use it to develop apps. We abstain from performing automatic (updates) patching in the background because this is too much control over developer’s source code. Further, it is not possible (with absolute reliability) to guarantee that the patching is free of unintended side effects. Additionally, developers should be informed and in control of the changes on their projects.

Up2Dep analyzes the developer’s code and provides developers information about the compatibility of the dependency’s update. In case an update to the latest version is incompatible, developers are provided with two options: either they can update to the latest compatible version without having to adjust their app’s code; or they can update to the latest version and Up2Dep provides them with information about which library APIs have changed and recommends changes to their existing app code. Additionally, Up2Dep leverages information about publicly disclosed vulnerabilities of libraries and detected cryptographic misuse in Android third-party libraries to warn developers against using insecure versions of dependencies.

Figure 1 illustrates how the different components of Up2Dep interact with each other. Using LibScout [7] and Cognicrypt [29], we feed Up2Dep the pre-analysis results consisting of API dependency analysis (from LibScout) and cryptographic API misuse analysis (from Cognicrypt). These pre-analyses results are bundled into offline databases. This allows Up2Dep to provide developers real-time feedback as it does not need to repeatedly analyze all version history of the third-party libraries that developers include into their applications, which might incur unnecessary performance overhead. More importantly, Up2Dep does not need to send developer’s code or all library information to its server as this would potentially threaten the privacy of developers and their code. After developers open a project in their Android Studio, Uses Analyzer will read the Android Source Code to analyze it for usages of the included third-party libraries. Whenever developers open a gradle build file (i.e., where dependencies are specified, see Appendix B), Remote Dependency Inspector will run its inspection to check for outdated library versions. Finally, Dependency Resolver takes the results of
Dependency Inspector and Usages Analyzer to compare them against the pre-analysis results to gather the following information:

- Are there newer versions of the included library?
- To which extent can the included library be upgraded, e.g., is there any incompatibility, where is the incompatibility, how can the app code be adjusted?
- Does the included library contain any security vulnerabilities, and does the developer’s code happen to use this potentially insecure code?

We will now describe each component of Up2Dep in details.

3.1 Analysis Tools

As mentioned above, Up2Dep collects information about third-party libraries using existing analysis tools.

3.1.1 LibScout. We provide developers information about the API of third-party libraries that they include in their apps. In particular, we notify developers if they can upgrade a library to the latest version or if the newer version would be API-incompatible with the existing app code. Hence, we need to analyze the library history to find out if any of the used library APIs have changed in newer versions of the library. When such changes occur, we provide developers with further information on how they can adapt their existing code so that it will be compatible to the newer version of the library. To this end, we leverage the open-source tool LibScout to produce API information for each library version in our dataset.

Library API database: The last version (2.3.2) of LibScout contains a dataset of around 250 libraries. In this work, we build on and extend the library database of LibScout. In particular, a library on a third-party repository would usually come with a descriptive file, e.g., pom.xml, and we analyze those files to discover transitive dependencies of the libraries in the LibScout database. A transitive dependency is another library on which the library included by the app developer depends. For instance, the Facebook Login Android SDK library version 4.40.0 declares three transitive dependencies in its pom.xml file: Android AppCompat Library V7, Facebook Core Android SDK, Facebook Common Android SDK. To obtain a list of popular third-party libraries that developers commonly include in their projects, we crawl the F-Droid repository [6] to extract libraries included in open source apps. In the end, we have a dataset of 1,878 libraries with their version history. We also extend LibScout’s list of publicly disclosed vulnerabilities of third-party libraries. As of July 2019, our list contains 10 libraries with a total of 97 vulnerable library versions.

Determining API Compatibility: To determine the API compatibility between any consecutive library versions, we use the API diff algorithm of LibScout that operates on two sets of public APIs api\textsubscript{old} and api\textsubscript{new}, where api\textsubscript{old} is the API set of the immediate predecessor version of api\textsubscript{new}. An API is presented by its signature that consists of package and class name as well as the list of argument and return type, e.g., example.com.ClassB.foobar()$\text{java.lang.String}$. If api\textsubscript{old} = api\textsubscript{new}, two versions are considered compatible. If api\textsubscript{old} $\nsubseteq$ api\textsubscript{new}, the newer version has added new APIs but did not remove or change any existing ones. This is also considered as compatible (backwards). However, if api\textsubscript{old} includes APIs that are not included in api\textsubscript{new}, a type analysis is conducted to check for compatible counterparts in api\textsubscript{new}. Compatible changes also include generalization of argument types, e.g., an argument with type String is replaced by its super type Object. Generalization on return types is normally not compatible and depends on the actual app code that uses the return value. If any of the api\textsubscript{old} is not found in the set of api\textsubscript{new}, we consider 2 versions incompatible.

3.1.2 CogniCrypt. We employ the static analysis component of CogniCrypt, namely CogniCrypt\_SAST, to discover insecure uses of cryptographic APIs within the libraries in our dataset. CogniCrypt\_SAST takes rules written in the CRYSL language, which define best-practice for secure use of cryptographic APIs, and analyzes Java applications to find any potential violations of the predefined rules.

We choose CogniCrypt instead of other tools, such as [18, 20, 43], because CogniCrypt and CRYSL are publicly available and provide the flexibility in defining cryptographic rules while other tools mostly provide hard-coded rules, which are not easy to extend. Besides, CogniCrypt provides more comprehensive rules that result in three times more identified cryptographic violations in comparison to previous work [20], and the analysis finishes on average in under three minutes per application. More importantly, CogniCrypt leverages several extensions [13, 31] of the program analysis framework Soot [45], which performs intra- and inter-procedural static analysis that gives CogniCrypt and CRYSL a high precision (88.95%) and recall (93.1%).

CogniCrypt’s rule set [30] includes 23 rules covering Java classes involving cryptographic key handling as well as digital signing. All rules are available on Github [5]. Beside the these rules, we have also written an additional rule for http (to check whether a library uses http instead of https to communicate with a server).

Cryptographic API misuse dataset: We apply CogniCrypt to our dataset consisting of 1,878 libraries. We are able to analyze 1,725 (91.9%) libraries. It took CogniCrypt more than 3 hours to
analyze the remaining 153 libraries and we terminated Cognicrypt when processing a library exceeded 3 hours. Among the 1,725 libraries, 238 (13.80%) contain at least one cryptographic API misuse, and 70 of those affected libraries (29.41%) have fixed/removed the cryptographic misuse in their later versions. This means that, developers could easily avoid such (vulnerable) cryptographic API misuse by upgrading their project’s dependencies to the latest version. Figure 2 lists the distribution of the cryptographic API misuse of the libraries in our dataset. The list is headed by MessageDigest (35.0% of the top 10 misuses). One of the reasons why MessageDigest has a significantly higher number of misuse is that to use it securely, developers (suggested by the Java Cryptography Architecture Standard) must apply a sequence of method calls, e.g., MessageDigest.getInstance(algorithmName) followed by MessageDigest.update(input), followed by MessageDigest.digest(), etc., combined with minimum required length for the offset of the update method. This does not seem trivial to follow. In general, for Java classes such as MessageDigest, SSLContext, and Cipher, developers need to specify an algorithm or a protocol to work with and library developers often use an algorithm or mode of encryption that is considered insecure, such as ECB mode for encryption, or MD5 or SHA-1 for hashing. This puts these classes of misuse among the most common cryptographic API misuses in third-party libraries. Further, we have found 20 cases where the libraries (spanning across 93 library versions) use http to communicate with remote servers.

3.2 Remote Dependency Inspector

Android Studio is built on JetBrains’ IntelliJ IDEA software. However, the major challenge is the implementation of an Android Studio extension for Up2Dep as it is not well supported and very few documentation is available. To learn how the internal system of Android Studio works, we have to manually read Android Studio source code and examine its APIs (e.g., dynamically run and test them) as well as use reflection to access its internal (private) API to enable the crucial functionality of Up2Dep. To effectively inspect an Android project’s dependency, we need to implement a custom code inspection. With the gradle build system, Android developers need to declare their project’s or module’s dependencies (libraries) to analyze varieties of Groovy code fragments. For every Groovy-CodeBlock, Up2Dep looks for a dependencies tag and iterates over all declared dependencies to extract group_id, artifact_id, and version string of each dependency (see Section B.1). Up2Dep then checks if the current dependency is available in our dataset (i.e., it checks if we have pre-analyzed this dependency and if the information about its APIs is available in our database). In case the dependency is available in our dataset, Up2Dep gathers all information about the current version up to the latest version, including information on whether a version has security vulnerabilities. The reason we do this is to not only detect the latest version, but also the latest compatible version in case an incompatibility with the app code occurs while helping developers avoid versions with known security vulnerabilities. At this point, Up2Dep knows if a dependency is outdated and which version is the latest one.

Database maintenance: To allow continuous maintenance of Up2Dep’s database we set a crawler up to run periodically to get new versions of the libraries in our database and subsequently apply Cognicrypt to analyze them for cryptographic API misuse, and LibScout to identify API compatibility between library versions. The updated database is retrieved automatically inside Android Studio to timely provide developers with updatability and security information about their included third-party libraries. For publicly disclosed vulnerabilities, we update our database manually.

3.3 Usages Analyzer

As we want to provide developers with information regarding a dependency’s compatibility and the use of libraries with potentially insecure usages of cryptographic APIs, we need to analyze the developers’ code. We built a code dependency analyzer that traverses through all Java and Kotlin files. When developers open a project in Android Studio, and the indexing process of Android Studio has completed, Up2Dep starts to analyze the project’s dependencies. We decided to wait for the indexing process to be done before analyzing code dependencies, because it significantly speeds up the analysis process as code files (including resources) have been transformed into a preferable representation, namely PsiTree, that allows faster processing. Each file corresponds to a PsiTree, and PsiTrees can depend on each other and can contain sub-PsiTrees. For every file (PsiTree), Up2Dep extracts its dependent PsiTree, and resolves the PsiTree to find out if it is associated with an external (foreign) code file. In case of a foreign PsiTree, Up2Dep checks with the Project-FileIndex class (provided by IntelliJ / Android Studio) to examine if the corresponding code file is in library classes or library source. As the ProjectFileIndex class contains information about all included libraries, Up2Dep can resolve a library class or a library source to find its library information (e.g., library name and version). Once the resolving process is completed, Up2Dep records any usages of the library, e.g., method call (including constructor), and saves them for later references. At the end of the process, Up2Dep has a complete dependency tree of source code files (Java and Kotlin) and their corresponding used libraries with details on which library methods the app is using. More specifically, the result of Usages Analyzer is a mapping of multiple pairs: code file (Java or Kotlin) and corresponding used library including API usages.

3.4 Dependency Resolver

The results from Remote Dependency Inspector and Usages Analyzer are fed to Dependency Resolver. For each included library, Dependency Resolver checks the library’s usages in the app code as reported by Usages Analyzer. At this point, Dependency Resolver has information on which APIs of the currently included libraries are used in the developer’s code. If Dependency Resolver finds that any of the used APIs of an outdated library is no longer available in the library’s latest version, it picks the library version that is newer than the current version but contains all the used APIs (newer compatible version). Using the information of publicly disclosed vulnerabilities of third-party libraries, Dependency Resolver checks
if the currently included library version has a known security vulnerability. Additionally, Dependency Resolver looks up each used library API to detect if the API leads to cryptographic API misuse in the library, and the details of the misuse.

From all those information, Dependency Resolver gives developers the following warnings and potential fixes in their build.gradle files. The dependency

- is outdated and can be updated to the latest version.
- is outdated and cannot be updated to the latest version.
- is outdated and has a known security vulnerability.
- potentially uses a cryptographic API insecurely.

We notify developers about the security and outdatedness of their project dependencies in the build.gradle file as this is the location where developers would manage their project dependencies. Besides, we also leverage the IDE functionality to allow developers to use Up2Dep in batch mode to analyze the whole project and see the analysis results in a separate window. In the following, we describe how Up2Dep notifies developers about the above declared problems.

**Outdated version can be updated to the latest version:** In this case, all the used APIs of the outdated library are also available in its latest version. Dependency Resolver suggests developers to update to the latest version as it will be compatible to the developer’s code (see Figure 11 in Appendix). Developers can apply the suggested fix by using the default short-cut of Android Studio or clicking on the default bulb icon to apply the recommended fix. When this quick-fix is applied, the outdated version string of the library declared in the build.gradle will be replaced by the latest version.

**Outdated version cannot be updated to the latest version:** When not all used APIs of an outdated library are available in the latest version (e.g., because the library developer removed or changed methods), Dependency Resolver suggests developers to update to a newer but compatible version. This means the newer version would not require changes to the app code to adapt to the library’s API changes. Similar to the previous fix, developers can apply it by using the default short-cut or clicking on the default bulb icon. When no compatible version is available and developers still want to update the outdated library to the latest but incompatible version, they are provided the option Show Dependencies (see Figure 12 in Appendix). The purpose of the Show Dependencies fix is to give developers feedback on how and where they can migrate their project’s dependencies to the latest versions (see Figure 10 in Appendix).

**Outdated library version with known security vulnerability:** When the included library contains a known security vulnerability, Dependency Resolver alerts developers with an error (in red color) with details on the vulnerability. Developers can further check the vulnerability in the attached link to our Up2Dep project website (see Figure 8 in Appendix). Since a known security vulnerability can be a serious problem for the host app or end-user, we use a red warning instead of a normal warning (in yellow color) to notify developers. In this case, developers can upgrade to the latest version that contains the security patches. When the latest library version is also vulnerable, developers are recommended to consider not using this library.

**Use of insecure cryptographic APIs:** Similar to known security vulnerabilities, if any used library method happens to insecurely use a cryptographic API, Dependency Resolver warns developers in form of errors against using this API (see Figure 9 in Appendix). In this case, Up2Dep suggests to developers to update to the latest version if the used APIs in the latest version do not contain cryptographic API misuse. If the latest version still has that problem, developers can use the Show dependencies option to examine the location and necessity of the used library method and decide whether or not they can remove the used method call, or switch to another library.

### 4 EVALUATION METHODOLOGY

Our goal is to find out if it is technically feasible for Up2Dep to support developers in keeping their project dependencies up-to-date and in avoiding library versions with security problems, e.g., how many outdated (including insecure) libraries Up2Dep has helped developers migrate to the latest versions and which security vulnerabilities it has fixed for developers. We further examine developers’ Up2Dep experience in an online survey. Different aspects of Up2Dep in interacting with developers — studying developers’ behavior upon learning about the security of an included library, how security warning messages can be customized, how can we keep the balance between developers being annoyed and being informed, how developer’s mental model evolves — are not in the scope of this paper, and left for future work. In the following, we first describe how we enable developers to evaluate Up2Dep in-the-wild. We then report how we advertised Up2Dep and delivered it to Android developers for evaluation.

![Figure 3: In context feedback dialog.](image)

To enable developers to evaluate Up2Dep, we leveraged the remote study platform of FixDroid [39] to setup and conduct our evaluation. We included telemetric features that record whether a suggested quick-fix was applied. We provided developers the Feedback in context (see Figure 3) option where they could send us feedback on whether the suggested fix worked as expected, if they needed more information on any warning, or on other issues they encountered. In our instruction, we strongly encouraged developers to provide us feedback so that we could make Up2Dep better, this was where they can help us to help them, i.e., making a free-to-use tool better for them. Developers were also provided the option to opt-out of our telemetric data collection in Up2Dep’s settings. Before developers downloaded Up2Dep we clearly inform developers on which information we gathered about their usage (on our project’s website and in Android Studio plugin repository description). Our goal in this step was to make sure they are well informed before they decide to install our plugin.

---

This feature is adopted from Lint tool.
4.1 Recruitment
After we advertised Up2Dep’s prototype at an Android developer conference, we used Twitter and email as communication channels to keep in contact with developers and to recruit further developers. After we released Up2Dep with complete features, we advertised our tool on different Android developer forums, Android developers groups on Facebook, and in a related lecture at our institution to invite experienced students, who are working on real (non-study-related) Android projects⁴, into using Up2Dep. Finally, we sent an invitation email to an Android development team, with which we already had contact before, to ask the team to try out Up2Dep.

We abstained from sending emails to the contact information harvested from Google Play apps, as done in prior studies [9, 19, 26, 39], since those studies had an extremely low response rate and such mass emails may be considered as harmful/spamming behavior that would create a negative view from developers toward studies conducted by researchers.

4.2 Ethical Concerns
This study has been approved by our institution’s ethics review board. All telemetric information is gathered anonymously—we do not know who the developer is—and we do not collect the developer’s code. Furthermore, we clearly explain on our website which information we gather and provide developers the option to opt-out of our telemetry data collection at any time. Finally, all data is sent to our server over a secured connection.

5 RESULTS
In this section, we present our evaluation results, which provide the answers to our research questions (RQ) stated in Section 1. This covers both telemetric data of developers who filled out our exit survey as well as of developers who are using Up2Dep but did not answer our survey. Our evaluation has lasted for 81 days, the results we report in the following are from within this duration. All data related to Up2Dep tutorial was excluded from our results⁵. Finally, we briefly compare Up2Dep with LibScout and Cognicrypt.

5.1 RQ1: Would it be technically feasible to support developers in keeping their project’s dependencies up-to-date?
From the telemetric data and answers to our online survey, we can see that developers have made use of Up2Dep to keep their project dependencies up-to-date. In particular, Up2Dep helped developers upgrade their project’s dependencies (N=116) to the latest version in 34 real projects. We describe the data as well as the feedback developers have provided in details in the following.

5.1.1 Telemetric Results. As we included telemetric features in Up2Dep, we are also able to gather telemetric data from developers who did not participate in our survey. Of 56 developers who are using Up2Dep, 30 (53.57%) have applied quickfixes (N=116) provided by Up2Dep to update their project’s dependencies—i.e., updated an outdated third party library to the latest/newer compatible version or examined a library’s API dependencies (34 projects).

Figure 4 shows the number of applied quickfixes per type. We can see that the majority of applied quickfixes are Update to the latest version. Besides, 27.59% of quick-fixes belong to Show dependencies meaning that developers have checked the API usages of the corresponding dependency. However, since we do not collect developer’s code, we do not know whether manual code change were performed to update the corresponding dependency.

5.1.2 Survey Results. Additionally, we asked developers to fill out an online survey. Among all 30 developers who have applied suggested quick-fixes in their projects, 22 of them (73.33%) have provided us feedback through the feedback dialog (Figure 3). On average developers have spent 19 minutes working with Up2Dep before giving us the first feedback. The results from the 60 in-situ feedbacks we received from 22 developers confirm that 80.0% of the proposed fixes worked, and developers found the warning/quickfix useful while only 4 proposed fixes did not work as expected. Figure 5 lists all feedback provided by developers. We also observe that 5.0% of the feedback indicates that the developer did not understand the warning message (I don’t get it). We manually examined the corresponding third-party libraries and found that their warnings were about cryptographic API misuse. This suggests that we need to make the warning message more developer-friendly, e.g., make it easier to understand (similar to other domains such as browser security warnings [10, 23]). As each feedback came together with the dependency for which developers had given feedback, we manually investigated the feedback that belongs to False positive and Other. We noticed that transitive dependencies might be the reason for such feedback. When a third-party library A depends itself on library B in version v₁ and developers use library B version v₂ in their app code, this means this project has now two versions (v₁ ≠ v₂) of the library

---

⁴Projects that are not related to their university studies/courses
⁵When Up2Dep recorded telemetry data, it computed a hash value of the project’s name. If developers used Up2Dep in a project that has the same hash value with one of our exemplary projects, such data was excluded from our results.
Compatible version check
Insecure version check
Crypto Misuse API check
Show dependency
90.91%
72.73%
72.73%
72.73%
9 of them are students (2 with at least 2 years of programming experience), and their age ranges from 18 to 30 years. Among 23 developers, 9 of them are students (2 with at least 2 years of programming experience, 7 with less than 1 year of programming experience) who got to know Up2Dep after we advertised it in a related lecture at our institution\(^8\).

Usability score: To assess the usability of Up2Dep in our survey, we used the SUS (System Usability Scale) [17]. A system with a SUS score of above 68 would be rated as above average. Up2Dep achieves a SUS score of 76.20, which is considered good in terms of usability [15].

Useful features: Of Up2Dep’s features, the Compatible version check was named most often (see Figure 6). This supports the results of a previous study [19] that showed that developers abstain from updating their project’s dependencies due to (fear of) incompatible updates.

5.2 RQ2: Could such a tool support have a tangible impact on the security and privacy of Android apps?

5.2.1 Fixed Security Problems. We observe that there are 4 instances of the okhttp3 v3.0.0 library in developers’ projects, which contains a known security vulnerability. okhttp3 v3.0.0 allows man-in-the-middle attackers to bypass certificate pinning by sending a certificate chain with a certificate from a non-pinned trusted CA and the pinned certificate. Zhang et al. showed that nearly 10% of the most popular apps on Google Play store still used such an insecure version for more than 1 year after the fixed version had been released [47]. In our study, those library versions were updated by developers with the support of Up2Dep to the latest, fixed version. Furthermore, there are 3 instances of an outdated version of the Glide library where developers used hash API without calling the complete sequence of function (see Section 3.1.2). Finally, one instance of okhttp3 v3.11.0 that misused a cryptographic API, and the developer in our study happened to re-use the correspond API of the library. This issue has been fixed in their latest, misuse-free version of the library. All in all, 6.89% of the outdated dependencies that Up2Dep has helped developers to migrate to their latest versions (8 out of 116) had security problems. Since we do not collect information of the developers’ projects (i.e., this may make developers skeptical to try Up2Dep), we therefore do not have information on the projects patched by Up2Dep. However, regardless of the project details, we consider this number non-negligible given the easy means that developers can employ to fix them. Therefore, by fixing projects containing these insecure library versions, Up2Dep directly benefits the security and privacy of Android apps.

5.2.2 Security Problems of Transitive Dependencies. From the feedback related to the False positive category, we learned that for a small number (2) of cases, the problem of transitive dependency would prevent developers from keeping their project’s dependencies up-to-date because of incompatibility. However, the current dependency management system of Gradle makes it hard for developers to be informed about what are the transitive dependencies of the manually declared dependencies as it automatically downloads sub-dependencies of a given dependency without developers easily noticing it. Developers can check the log console to see what sub-dependencies are downloaded together with the current dependency, yet this is only available in the log console with hundreds of log events. The problem becomes more serious if a transitive dependency has (well known) security problems. Those are totally hidden from developers because they are usually automatically downloaded following the main dependency unless developers specifically exclude them [27]. Thus, even if developers would vet a dependency manually, insecure sub-dependencies that are automatically, non-obviously pulled in when installing the dependency can undermine the app’s security again. This highlights the need for tooing support, as such Up2Dep.

Transitive dependency analysis: Given the crucial information regarding security problems of transitive dependencies, we

\(^8\)We did not distinguish academic training from programming experience.
developed an additional feature that thoroughly checks all transi-
tive dependencies of all declared dependencies to: (1) analyze
compatibility when suggesting developers to update the declared
dependencies, and more importantly, (2) to check and notify de-
velopers if any transitive dependencies contain security problems.
When Up2Dep detects a declared dependency in a build.gradle file
of a project, it checks all transitive dependencies (all sub levels)
of the current dependency and queries security related informa-
tion of these transitive dependencies. If any transitive dependency
contains security problems, developers are notified similar to how
security problems of the main dependency are communicated (see
Section 3.4).

Analysis Results: Our results reveal that there are 1,209 library
versions (belonging to 112 unique dependencies) that have security
problems. These dependencies are currently (transitively) used by
9,787 library versions (233 unique libraries) in our data-set. Espe-
cially, among 1,209 transitive dependency versions with security
problems, 16 contain a publicly disclosed vulnerability. This means
even if developers are aware of such libraries with security prob-
lems they have no way to find out if their projects are including
such insecure dependencies as they are not visible to developers.

The latest version of Up2Dep now informs developers about such
security problems of both the main dependency and transitive de-
dendencies so that developers can also avoid insecure transitively
included library (versions).

5.3 Comparison with Existing Work

In our work, we significantly increased the database of LibScout
by a factor of 7.5x. Furthermore, our database covers the top 100
most popular libraries on Maven repository [4] which was not
considered by LibScout. Most importantly, we provided an effortless
synchronization (end-to-end) process that automatically scans for
new libraries (versions), analyzes for cryptographic API misuse,
then the information on security and updatability of new libraries
(versions) are delivered to developers right in their development
environment without them having to use extra tools.

Besides, as we extended the rule set of Cognicrypt to include the
check for use of http protocol, we have found 20 libraries (8.4% of
all identified insecure libraries), spanning across 93 versions using
such insecure protocol. With the original rule-set of Cognicrypt
we would have missed the insecure network connection in these
libraries.

6 DISCUSSION

6.1 Threats to Validity and Future Work

Our work leverages LibScout and Cognicrypt and inherits their lim-
itations. For LibScout the ability to provide suggestions for API
changes relies solely on API heuristics, such as name, parameter
types, or return types, which do not necessarily guarantee that the
suggested API will work as expected. If the semantics or side-effects
of a library method change between versions, this could break the
functionality of the developer’s app although the app code was
compatible at the method signature level with the new library ver-
sion. Further detecting semantic changes is an open problem that
requires effort from different domains, especially software engineer-
ing, and is not in the scope of our work. Yet in this work, we show
that relying on API changes to derive compatibility among library
versions does help developers to keep their project’s dependencies
up-to-date, yet it needs further improvement to cover more cases.

While Cognicrypt provides the flexibility to create new rules to
detect cryptographic misuse, it is not free of false positives. We
found cases where calls to cryptographic APIs are wrapped in
custom Java utility classes by the library developer. Cognicrypt
can not completely link the control flow graph of those custom classes
to detect if a cryptographic misuse occurs in those cases. This
results in Cognicrypt over-approximating the misuse and reporting
false positives. In particular, misuse of MessageDigest depends on
call sequences and this shortcoming of Cognicrypt in classifying
misuse of that class when being wrapped in custom classes might
be a contributing factor to the high number of misuses detected for
MessageDigest (see Figure 2). However, it is not easily possible to
verify such misuse using static analysis and exclude false positives
from our results. Once Cognicrypt addresses this limitation, also
Up2Dep will provide more accurate warnings to app developers.

Additionally, we currently manually look for publicly disclosed
vulnerabilities, which is a tedious task. In future, this could be gen-
erally done with a central library repository, e.g., when a vulnerabil-
ity of a library is disclosed, central library repositories can incorporate
and mark the vulnerable versions in their database so that tools
like Up2Dep or Lint can automatically retrieve and provide devel-
opers feedback in their IDE. However, for the cryptographic API
misuse, Up2Dep’s pre-analyzer component automatically crawls
newer versions of third-party libraries and runs Cognicrypt to ob-
tain up-to-date results.

Further, the population size of the developers in our evaluation
might be perceived as small since we only have 56 developers, of
which 23 shared with us their experience in our online survey, and
22 developers provided us feedback in their Android Studio. Our
demographic data shows that our evaluation indeed has a popula-
tion of experienced developers (e.g., 18 of them have developed at
least 2 Android apps). However, developer studies [9, 26, 39] had in
the past notoriously a low number of participants as it is not easy
to recruit real developers. Besides, most of them were conducted
with students as proxies using handcrafted, toy projects which do
not necessarily represent the day-to-day real situation that devel-
opers often face. In our work, on the other hand, we tried to avoid
students as proxies and toy projects as much as possible and gain
insights from developing real app projects (external validity). We
think the fact that we could recruit this number of developers and
keep them using Up2Dep is in part due to the interest and need
for such a tool by the developer community. Furthermore, with
our feedback in-context option, we obtain valuable feedback from
developers on whether Up2Dep works. Given the only small per-
centage of false positives reported (6.7%) and 80% of the suggested
quick-fixes working as expected, we believe that we have delivered
a novel and expedient tool, and can show the impact of such tooling
support on real world situations.

Lastly, we abstained from collecting telemetric information on
whether developers ignored the quick-fix, since this might be con-
sidered too intrusive. Unfortunately, this also precludes us from
modeling whether a known security vulnerability or cryptographic
misuse warning is a significant predictor for applying quick-fixes
and library updates in our evaluation.
6.2 Transitive Dependencies and App Security

While during our evaluation, we did not consider transitive dependencies, we also have seen that the problems of transitive dependency with regards to library upgradability is a corner case, e.g., only 2 instances of the false positives. Also existing research [28] on the upgradability of third-party libraries shows that only 1.7% of the library API could be affected by this problem (referred to as entangled dependencies). Still we see a potential threat to the security of Android apps due to transitive dependencies. We found that (known) security problems of a library could be hidden from developers when the library is included as a transitive dependency of another dependency and this transitive dependency is not communicated as obvious to app developers as needed. While the community is trying its best to find security related problems of third-party libraries, it is also important to keep developers informed on all potential risks associated with a (declared) dependency. We are to the best of our knowledge the first to study the security problems of transitive dependencies and subsequently developed a solution to tackle this problem by alerting developers when they include libraries that have transitive dependencies with security problems.

6.3 Impact of Fixing Insecure Dependencies

Among the 116 applied fixes, 6.89% had security vulnerabilities (4 known security vulnerabilities, 4 cryptographic API misuse). We consider these numbers non-negligible and this has tangible impact on the security and privacy of the Android apps that developers are working on. Previous work has identified the security & privacy impact of outdated third-party libraries in general and of outdated insecure third-party libraries in particular (see Section 2). By updating the insecure code dependencies to secure versions, we are removing the factors that could amplify security & privacy problems in apps and expose end users to multiple types of attacks. While market stores such as Google Play have been scanning apps for security & privacy problems, they are dealing with monolithic byteblobs where there is no separation between app code and library code. Hence, such solutions need exact, reliable library detection mechanisms which is a challenging task and no satisfactory solution exists yet. This becomes even more challenging when the app’s byte-code is obfuscated, something that Google itself is promoting to app developers [12]. Our results show that by integrating support to suggest secure code dependencies within developers’ IDEs, we can eliminate many security problems that arise from including insecure third-party libraries without having to deal with monolithic apks where app code and library code have been merged together. Especially, developers do not need to learn new tools or adjust their daily work-flow to be able to use Up2Dep. Our results call for action from IDE developers to merge tools like Up2Dep into IDEs, like Android Studio, so that developers immediately and by default benefit from such support. Based on our results, the experiences in other software ecosystems [8, 24] or for native Android libs [11, 25], and the movement toward integrating security into software development life cycle namely SevDevOps [36, 37], we argue that this would have a tangible impact on the security & privacy of the Android ecosystem especially.

6.4 Fear of Incompatibility vs. Will to Update

In our evaluation, we learned that, the majority of the outdated libraries can be updated all the way to the latest version (see Figure 4) without having to change the app code (i.e., 68.97% quick-fixes are update to the latest version). Developers are afraid of updating because they fear that the new version of the libraries would break the app’s existing functionality [19]. Without the information on the compatibility of the new update, developers either have to manually verify the release notes (if available) of the libraries to make sure that the functions their apps are using are still available in the update, or simply keep using the outdated versions. One developer shared such experience via email with us after trying out Up2Dep:

“Thank you for sharing your project with me. It’s really exciting, we’re normally manually reviewing the change logs to see if we should update our dependencies right away or what we should test.”

Compatible check was rated the most useful feature (see Figure 6) by developers in our study. Had Up2Dep not provided the compatibility information on the outdated dependencies, developers would probably not be willing to perform the updates on these 68.98% outdated dependencies (80 of 116 outdated dependencies).

7 CONCLUSION

Since security patches of libraries are often rolled out as updates, app developers (not market stores, not the end users) need to keep their project’s third-party libraries up-to-date to avoid security problems of outdated libraries. In this paper, we present Up2Dep, an Android Studio extension that facilitates the task of keeping an Android app project third-party libraries up-to-date while taking into account the security and the compatibility of the newer versions of such dependencies. Up2Dep suggests alternative library APIs to developers in case a newer library version is incompatible. It further helps developers in avoiding insecure libraries by alerting them to publicly disclosed vulnerabilities and cryptographic API misuse in third-party libraries. We tested Up2Dep with 56 Android developers. Up2Dep has helped developers in fixing 116 outdated third-party libraries, of which 6.89% had security vulnerabilities (4 known security vulnerabilities, 4 cryptographic API misuse). The majority (80.0%) of the suggested quick-fixes worked as expected with only 4 cases of failed quick-fixes. In further investigation, we discovered the hidden security problems of transitive dependencies of 13.51% of the libraries in our dataset. We are the first to discover the hidden problem of insecure transitive dependencies and subsequently developed the corresponding solution to tackle this problem. Our results call for action to (1) merge tool support, like Up2Dep, into developers’ integrated development environments, as this would create a tangible impact on the security and privacy of the Android ecosystem when developers benefit from tool support for upgrading used third-party libraries, and (2) study developer’s behavior to best provide them the right tool support.

ACKNOWLEDGMENTS

We like to thank our anonymous reviewers for their valuable feedback. We also thank our shepherd — Sébastien Bardin for his feedback and for being a supportive point of contact for the revision of this paper.
A SURVEY QUESTIONS

A.1 App Development

Q1: How do you prefer getting update notifications? [multiple choice]
   • Yellow highlighting on the dependency version
   • Pop up when new versions are available, with "Ignore" option
   • When I build/compile my project?
   • Other [free text]

Q2: Based on which criteria do you usually pick a library for your projects? [multiple choice]
   • Popularity
   • Easy to use
   • Functionality
   • Security
   • Other

Q3: Have you developed any third-party libraries? [Yes/No]
   • Yes: Which library is that? [free text]
   • No

Q4: How would you rate the security (whether a given version has security vulnerability) of libraries you decide to include it into your projects [single choice]
   1-5

Q5: Did you notice any highlights regarding outdated library versions in your app’s Gradle files? [single choice]
   • Yes
   • No
   • I don’t know

Q6: Where do you reach out for help while solving programming tasks that relate to third-party libraries? [multiple choice]
   • StackOverflow
   • Search engines
   • Third party library’s website
   • Other [free text]

A.2 Up2Dep Usage

Q7: How did you get to know Up2Dep? [multiple choice]
   • Friends, colleagues
   • IntelliJ IDEA/Android Studio repository
   • Twitter
   • Android Developer Conference
   • Other

Q8: Which features of Up2Dep do you find useful? (screenshots are included for each feature)
   • Compatibility check (compatible version vs. latest version)
   • Insecure version check
   • Crypto API misuse check
   • Show dependencies and alternative API suggestions
   • Other [free text]

Q9: Since you started using Up2Dep, how many outdated libraries have you updated?
   • 0
   • 1
   • More than 2
   • Other [free text]

A.3 Up2Dep Usability - SUS Questions

Q10: For each of the following statements, how strongly do you agree or disagree (Strongly disagree, disagree, neutral, agree, strongly agree)

Q11: How many years have you been programming in Android?
   • less than 1 year
   • around 2 years
   • around 3 years
   • more than 3 years

Q12: How old are you?
   • 18–30
   • 31-40
   • 41-50
   • >50
   • No answer

Q13: What is your gender?
   • Male
   • Female
   • No answer

Q14: How many apps have you developed so far?
   • 1
   • 2
   • more than 2
   • 0

Q15: Do you have IT-Security background?
   • Yes
   • No

Q16: Where are you from? [free text]
We will briefly provide information on the Gradle build system and we describe how code inspection and quick-fixes work in IntelliJ Android Studio is based on JetBrains IntelliJ IDEA. Therefore, to Android Studio uses Gradle Build Tool as an Android Studio via central repositories. Besides, for local module dependencies and projects.

Listing 1: Declaring external dependencies in Android projects.

B BACKGROUND
We will briefly provide information on the Gradle build system and Android Studio plugin development.

B.1 Gradle Build Tool in Android Studio
Android Studio uses Gradle Build Tool [2] as an Android Studio plugin to automate and to manage the app build process. The Gradle build system eases the task of including internal and/or external libraries to app builds as dependencies. In our work, we do not take into account local binary dependencies, e.g., jar files that developers manually download and import into their projects because the majority of third-party libraries are included in Android projects via central repositories. Besides, for local module dependencies and local binary dependencies, the exact version information is not available, one can only profile the binary files and provide approximate matches which would add up another factor of uncertainty.

Listing 1 shows examples of how developers can declare their project’s external dependencies in Android Studio. On line 3, components of a dependency’s information are colon-separated, group_id:artifact_id:version, while on line 5, they are declared as key-values. From this information, when developers choose to sync their project’s dependencies, Gradle will sync such dependencies from the default repository (e.g., JCenter or Maven) or the ones declared in the gradle.settings file of the app project. Besides, developers can also declare version strings as a variable (line 1) and use this variable for the external dependency’s version (lines 7,8). This helps developers avoiding repeatedly specifying (and updating) version strings for multiple libraries from the same group (e.g., com.android.support) that use the same version string.

B.2 Android Studio Plugin
Android Studio is based on JetBrains IntelliJ IDEA. Therefore, to develop an Android Studio plugin one needs to create an IntelliJ IDEA plugin that targets Android Studio. The IntelliJ platform provides tools designed for static code analysis, i.e., inspections that allow developers to check for potential problems in the source code. Examples of such inspections are finding probable bugs, dead code, performance issues, improving code structure and quality, or examining coding practices and guidelines. In the following, we describe how code inspection and quick-fixes work in IntelliJ IDEA/Android Studio. Code inspection in Android Studio leverages the program structure interface (PSI) to analyze source code files of a project. PSI is responsible for parsing files and creating syntactic as well as semantic code models. This allows the IDE to efficiently perform static code analysis on a project’s source code such as identifying code inconsistency, probable bugs, and specification violations. There are two main program structure interfaces in IntelliJ IDEA namely PsiFile and PsiElement. PsiFile represents the content of a code file as a hierarchy of elements (so-called PsiTree). Each specific programming language can extend the PsiFile base class to have its own representation, such as PsiJavaFile for Java language, GroovyFileBase for Groovy language, or KtFile for Kotlin language. PsiElements are used to explore the internal structure of a project’s source code by the IntelliJ platform.

Specifically, PsiElements are used to perform code inspection and quick-fixes on IntelliJ IDEA/Android Studio projects. When a quick-fix is applied, PsiElements are updated, removed from, or additionally added to an existing PsiFile.

To analyze developer’s code, one can extend the InspectionProfileEntry class to build a PsiElementVisitor that traverses through all PsiElements belonging to a PsiFile. Each PsiElement corresponds to a keyword, a variable, or an operation in a particular language. To apply a quick-fix, e.g., updating a dependency declared in the file build.gradle of an Android project (see Appendix B.1), a new PsiElement representing a newer version is created and replaces the existing PsiElement that represents the outdated library’s version.

Figure 7: Invitation to our online survey inside Android Studio.

Table 1: Participant demographics of online survey.

<table>
<thead>
<tr>
<th></th>
<th>Age</th>
<th>Gender</th>
<th>Based</th>
<th>Programming Experience (years)</th>
<th>Apps Developed</th>
<th>IT-Security Background</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>18-30</td>
<td>Male</td>
<td>Europe</td>
<td>&lt;1</td>
<td>&gt;2</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td></td>
<td>Asia</td>
<td>2</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Other</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&gt;3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>


Figure 8: **Up2Dep** warns against using an insecure library version (with publicly disclosed vulnerability).

Figure 9: **Up2Dep** warns against re-using a cryptographic API misuse in a library.

Figure 10: **Up2Dep** shows how developers can migrate their project dependencies to the latest version when incompatibility between library versions occurs, i.e., the return type of method `load` has changed from `RequestBuilder` to `RequestCreator`.

Figure 11: **Up2Dep** warns against an outdated library.

Figure 12: **Up2Dep** provides different options to update an outdated library version.