AndroLibZoo: A Reliable Dataset of Libraries Based on Software Dependency Analysis

Jordan Samhi[α], Tegawendé F. Bissyandé[β], Jacques Klein[β]
[α]CISPA – Helmholtz Center for Information Security, jordan.samhi@cispa.de
[β]SnT, University of Luxembourg, Luxembourg, {tegawende.bissyande, jacques.klein}@uni.lu

ABSTRACT

Android app developers extensively employ code reuse, integrating many third-party libraries into their apps. While such integration is practical for developers, it can be challenging for static analyzers to achieve scalability and precision when libraries account for a large part of the code. As a direct consequence, it is common practice in the literature to consider developer code only during static analysis –with the assumption that the sought issues are in developer code rather than the libraries. However, analysts need to distinguish between library and developer code. Currently, many static analyses rely on white lists of libraries. However, these white lists are unreliable, inaccurate, and largely non-comprehensive.

In this paper, we propose a new approach to address the lack of comprehensive and automated solutions for the production of accurate and "always up to date" sets of libraries. First, we demonstrate the continued need for a white list of libraries. Second, we propose an automated approach to produce an accurate and up-to-date set of third-party libraries in the form of a dataset called AndroLibZoo. Our dataset, which we make available to the community, contains to date 34 813 libraries and is meant to evolve.

ACM Reference Format:
Jordan Samhi[α], Tegawendé F. Bissyandé[β], Jacques Klein[β]. 2024. AndroLibZoo: A Reliable Dataset of Libraries Based on Software Dependency Analysis. In *21st International Conference on Mining Software Repositories (MSR '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3643991.3644866

1 INTRODUCTION

Static analysis is a popular technique used in the literature to analyze Android apps, it analyzes app code without executing it. This approach is widely used to uncover security issues [17]. For example, researchers apply static analysis techniques to detect privacy leaks [3, 15, 21, 22, 26] and detect malicious code [8, 10, 23, 24]. However, most of these approaches need to differentiate between developer and library code to focus on the app's functionality, which is the relevant code for finding security problems and avoiding scalability issues (due to the widespread use of polymorphism in libraries and the over-approximation of static analyzers). This is why static analyzers do not dive into the Android framework code during analyses (e.g., FlowDroid discards classes that are within the Android framework, cf. lines 64–69 in FlowDroid's SystemClassHandler class [20]). Differentiating between developer and library code is, therefore, a crucial step for static analysis to be effective and more scalable [27, 29], as it allows analyzers to focus on the parts of the app that are most likely to contain security issues.

Furthermore, libraries can introduce noise for malware detection. For example, Mudflow [4] and DroidAPIMiner [1] show that discarding libraries in their analyses improves their malware detection performance. A reliable list of libraries is thus an important artifact for the research and analyst community.

Previous studies have employed white lists to identify libraries in Android apps. Chen et al. [9] manually compiled a list of 73 package names from common libraries. Similarly, Grace et al. [11] randomly selected apps from a dataset of 100 000 apps that were manually screened to identify libraries. With this approach, they created a list of 100 libraries. These lists are ad-hoc and incomplete, as they only contain 73 and 100 libraries, respectively. Another method to build a white list of libraries has been proposed by Li et al. [16]. Their approach involves using a large dataset of apps to identify candidate libraries. The process includes ranking all package names by frequency of appearance in apps and using heuristics to retain libraries. However, though the approach is considered the *state-of-the-art* white list of libraries in the literature [6], the list provided is outdated, the method used to create the list relies on arbitrary heuristics, and the hypothesis to consider a package name as a library according to its occurrence leads to a high rate of false positives (if numerous versions of the same app are present, for instance: 20 715 different versions of app "com.slideme.sam.manager" are present in AndroZoo). Hence, creating a comprehensive white list of libraries to discriminate the developer code from libraries accurately remains an open challenge.

In this work, we propose a novel approach to build the first extensive and precise, by construction, white list of libraries by mining software dependencies. Contrary to the research literature, which often relies upon complex approaches [28], our method involves mining information from developer habits. We propose to the research and analyst communities a dataset called AndroLibZoo, containing 34 813 libraries. AndroLibZoo is accurate by construction, i.e., it only contains libraries. This dataset is meant to evolve and regularly incorporate new libraries added by third-party vendors. AndroLibZoo aims to facilitate the work of static analysis analysts in terms of scalability and robustness. We believe this dataset will be a valuable resource for static analysis, we encourage its use and expansion as new libraries become available.

Overall, this paper makes the following contributions:

- We show that white list of libraries are still needed.

MSR '24, April 15–16, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0587-8/24/04…$15.00
https://doi.org/10.1145/3643991.3644866

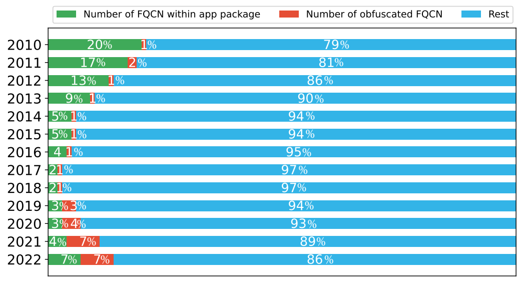

Figure 1: Proportion of FQCNs within the apps' package and obfuscated FQCNs

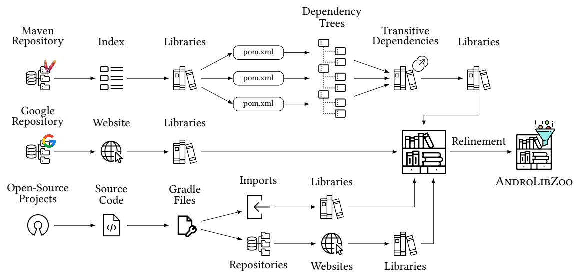

Figure 2: Overview of our methodology to construct AndroLibZoo, a collection of third-party libraries.

- We propose an approach to build an accurate set of libraries.
- We build AndroLibZoo, the first version of the library dataset produced by our approach.

Our artifacts are available: https://github.com/JordanSamhi/AndroLibZoo, https://zenodo.org/records/10072709

2 MOTIVATION

In this section, we motivate our work with a study to demonstrate the need for a white list of libraries.

Despite existing approaches' limitations, it is unclear whether white lists of libraries are still necessary in practice, mostly due to the usage of obfuscation. Hence, a legitimate question is: **To what extent does obfuscation jeopardize the use of a white list in Android apps?** To address this question, we conducted a motivating study examining the prevalence of package name obfuscation in Android apps.

Let us first introduce name obfuscation, a technique used to remove package names and/or class names' meaning to prevent or hinder reverse engineering and make malicious code detection more difficult. Typically, package and class names have meaningful names to make it easier for developers to understand the app structure and the code's purpose. Obfuscation is both used to protect apps' intellectual property and to make malicious code harder to detect. Most code obfuscators replace package or class names with simple letters from the alphabet [16, 18]. For instance, the package name "com.example.myapp.MyClass" could be obfuscated such as: ① only the class name could be ofuscated: `com.example.myapp.a`; ② only the package name could be obfuscated: `a.b.c.MyClass`; ③ both the package name and the class name can be obfuscated: `a.b.c.d.a`; or ④ the package name can be removed: `a`.

To conduct this study, we randomly selected 10 000 apps per year from the Androzoo [2] dataset for each year from 2010 to 2022 and checked whether any of the Fully Qualified Class Names (FQCNs) in these apps either: ① started with a letter of the alphabet (e.g., `a.b.*`, or `g.u.*`); or ② its class name is a single letter of the alphabet (e.g., `com.example.a` or `f.w.i`). This allows us to cover all the cases cited above.

The results of this study are shown in Figure 1. This figure presents three categories of FQCNs: ① the number of fully qualified class names that are within the package name of the app, i.e., the one declared in the manifest (e.g., if the package name of the app is `org.example` and FQCN is `org.example.MyClass`); ② the number of FQCNs that have been obfuscated, as defined previously; ③ the number of all remaining FQCNs, i.e., FQCNs that are not within the package name of the apps nor in an obfuscated package name.

This study shows that a low percentage of classes are in the app package (i.e., package with the app package name). But more importantly, the percentage of obfuscated FQCNs in the apps is even lower, even if this percentage appears to be growing over time. The largest percentage is made up of FQCNs that are neither within the package name of the app nor obfuscated. These results suggest that the use of name obfuscation techniques is not yet widespread enough to render white lists unnecessary for static analysis of Android apps. Therefore, it appears that white lists are still useful for differentiating these FQCNs from developer and library code.

3 DATASET CONSTRUCTION METHODOLOGY

This section presents our methodology to build AndroLibZoo.

Hypotheses: Our hypothesis is that: ① the majority of Android app developers rely on Android Studio and Gradle to build their apps; ② that Gradle in turn relies on Maven and Google to retrieve libraries [12, 13]; and ③ open-source Android apps can bring valuable information about the library used in apps, thanks to the availability of configuration files.

Based on these hypotheses, we set up a straightforward approach that can be seen in Figure 2. Our approach involves two main steps: ① extracting libraries from the Maven and Google's Android library repositories; and ② extracting libraries used in open-source projects. Maven is a widely used tool for managing dependencies and building Java-based projects, and the Maven repository is a central location where developers can find and share libraries. Google's Android library repository is a collection of Android libraries available through the Google Maven repository. In the following sections, we explain how we proceed to build AndroLibZoo.

3.1 Mining repositories

This section details how we obtained the first portion of our dataset by leveraging public library repositories. Our process is divided into two steps: ① obtaining package names from the repositories. ② gathering transitive dependencies from the package names.

3.1.1 Package Name Collection. This section explains how we collect package names from Maven and Google repositories.

Maven. First, we download the Maven index file (updated weekly) and the Maven Indexer Command Line Interface (CLI) tool which

```
com.android.tools:sdk-common:22.9.0
\- com.android.tools:sdklib:22.9.0
   +- com.android.tools:dvlib:22.9.0
   |  \- com.android.tools:common:22.9.0
   |     \- com.google.guava:guava:15.0
   +- org.apache.httpcomponents:httpclient:4.1.1
   |  +- org.apache.httpcomponents:httpcore:4.1
   |  +- commons-logging:commons-logging:1.1.1
   |  \- commons-codec:commons-codec:1.4
   +- com.android.tools.layoutlib:layoutlib-api:22.9.0
   |  \- net.sf.kxml:kxml2:2.3.0
   +- org.apache.httpcomponents:httpmime:4.1
   \- org.apache.commons:commons-compress:1.8.1
```

Figure 3: Dependency tree of the com.android.tools.sdk-common library version 22.9.0.

are available publicly. The Maven index is a file created using the Lucene library [7] and contains metadata about the artifacts available in the Maven repository. The Maven Indexer CLI is a tool used to extract this file. Once the Maven index file is extracted, we use a Java program of our own, called SearchLuceneIndex, to extract all the *groupIds* from the index. At the time of writing, our list from the Maven repository contains a total of 69 316 entries.

Google. We could extract a total of 235 libraries by accessing the Maven Google repository. Once we have obtained the package names from both the Maven repository and the Maven Google repository, we merge the two lists into a single list, eliminating any duplicate entries. This results in a list of 69 535 package names.

3.1.2 Transitive Dependency Extraction. Libraries often rely on other libraries known as *transitive dependencies*. For example, the `com.android.tools.sdk-common` library in version 22.9.0 has 12 transitive dependencies. Figure 3 shows the dependency tree of this library. To create a comprehensive and up-to-date set of libraries, it is important to consider these transitive dependencies.

We used the *mvn* utility (from Maven) to build a given project's dependency tree. However, since our dataset is made of libraries (i.e., package names) and not actual development projects, getting the dependency trees from the libraries is not trivial. We implemented the following strategy. First, from the maven index, we obtained all artifacts in the form of the artifactIds (e.g., com.example) concatenated with groupIds (e.g., MyLibrary) and versions (e.g., 1.0) to get the real library names, resulting in a list of 10 069 375 unique libraries. We considered all available versions for each library because transitive dependencies can differ from one version of a library to another. Then, for each library, we generated a `pom.xml` file with a dummy project and the library as a dependency of the project. We then launched the `mvn dependency:tree` command to get the list of all transitive dependencies of the library. Obtaining transitive dependencies for all libraries in our dataset is computationally intensive. With a total of 10 069 375 libraries, extracting transitive dependencies for each library would have taken an unreasonable amount of time. Therefore, we set a timeout of 30 seconds for each call to the `mvn dependency:tree` command. Even with this timeout, 9 312 664 (i.e., 92.5%) dependency trees were successfully built, and the process still required 21 days of computation using 70 instances in parallel on a Debian server with an AMD EPYC 7552 48-Core Processor CPU with 96 cores and 630GB of RAM. Then, we parsed the dependency trees generated and built a list of 15 089 additional libraries with any duplicate removed. It should be noted that this number (i.e., 15 089) was reached long before (after only 10 days) the 9 312 664 dependency trees were built. Finally, we merged this list with the one generated previously, and we removed any duplicates, resulting in a final list of 69 877 package names.

3.2 Mining open-source Android projects

Open-source projects provide valuable information about the libraries, relying on various third-party libraries to provide specific features and functionality. In the following, we describe the process for extracting lists of libraries from open-source projects.

We first obtained a list of open-source Android projects by downloading all apps in AndroZoo that were collected from the F-Droid repository. At the time of writing, this represented 4464 apps. We then used the F-Droid website to crawl the source code links for these apps and attempted to clone the repositories. Out of the 4464 apps, we could successfully clone 3425 projects (some reasons why some repositories cannot be cloned involve repositories that do not exist anymore, or the "git clone" command cannot work for, e.g., svn repositories, our prototype currently ignores non-git repositories). Next, we searched the `build.gradle` files available in these projects and extracted information about the libraries used. Specifically, we searched for the Gradle commands `implementation`, `classpath`, and `compile`, commonly used to declare dependencies. After parsing the `build.gradle` files and extracting the relevant information, we obtained a list of 463 unique libraries, which brings our total count of libraries to 69 980 after deduplicating.

To obtain a list of libraries as comprehensive as possible, it is not enough to consider libraries explicitly imported in the `build.gradle` files of the app projects. It is also necessary to consider library repositories (other than the default Maven and Google) declared in the `build.gradle` files and use this information to search for additional libraries. This process led to two additional repositories: jcenter and gradlePluginPortal. However, we found jcenter is deprecated and all its libraries are now in the Maven repository [14], which we have already mined. The gradlePluginPortal website proved to be a valuable resource. We crawled the gradlePluginPortal website, comprising 656 pages at the time of writing, and obtained an additional list of all package names available. This led to the discovery of 6515 libraries, which we added to our list after removing potential duplicates and led to a total of 76 007 libraries.

3.3 Refinement

We have designed a refinement process to remove unnecessary package names from the list of third-party libraries. This process helps ensure that the list is as concise as possible. For example, if the list contains the package names “com.example.subpackage" and “com.example", we would only keep “com.example" because this would be sufficient for identifying “com.example.subpackage.MyClass" as a library in an app. This is done by iterating over the list of package names and checking whether any element e_1 in the list starts with another element e_2 in the same list. If this is the case, the element e_1 is removed from the list. This process is repeated until no more changes are made to the list. Overall, our refinement process yields a list of 34 813 package names by discarding 41 194 of them (i.e., 76 007 − 34 813). Table 1 shows the details of all the steps of our methodology with the number of libraries collected.

Table 1: # of libraries after each step (OS = open-source)

Source					
Maven	69 316				
Google	+235 → 69 535				
Transitive Dep.		+15 089 → 69 877			
OS imports			+463 → 69 980		
gradlePluginPortal				+6515 → 76 007	
After Refinement					34 813

each resulting number is given after removing duplicates

Table 2: Number of package names per field in AndroLibZoo

Fields	Count	Fields	Count
with 1 field	732	with 6 fields	134
with 2 fields	17 172	with 7 fields	40
with 3 fields	13 086	with 8 fields	26
with 4 fields	2979	with 9 fields	8
with 5 fields	634	with 10 fields	2
Total			34 813

4 DATASET DESCRIPTION

In this section, we describe our dataset. AndroLibZoo contains 1210 unique roots. A root refers to the first element in a package name. For example, in "com.example.mypackage", "com" is the root.

We also collected data on the number of fields in the apps' package names in our dataset. A field refers to a dot-separated element in a package name. For example, in "com.example.mypackage", there are three fields: "com", "example", and "mypackage". Results are visible in Table 2. There are 732 package names with only one field, 17 172 package names with two fields, etc.

There are significantly fewer package names with four or more fields. The number of package names with one field is relatively low compared to the others. This suggests that many package names in the dataset follow a standard naming convention with a domain name followed by one or more subpackages. The presence of package names with four or more fields may indicate the use of more complex or specialized naming conventions.

Table 3 presents the top 10 most frequent roots and the top 10 most frequent fields. In the first two columns, we see that "com" is by far the most frequent root, with more than 13 000 occurrences. The second most frequent root is "org", with 5450 occurrences. In the second two columns, representing the most frequent fields, including the roots, we see that "com" field is still the most frequent. The second two columns do not differ much from the first two columns, except for the field "gradle" that now appears. This could indicate that Android libraries are prevalent (often built with gradle). It is confirmed in the last two columns, representing the most frequent fields without the roots, in which we see that "gradle", and "android" fields are the most frequent, with 680 and 443 occurrences respectively. After "gradle" and "android", the third most frequent field is "maven", with 352 occurrences. We see a shift in the most prevalent fields. Instead of roots, we now see fields such as "sdk", "maven", "plugin(s)", "api", and "tools". This may be indicative of the types of libraries. Overall, the results suggest that most libraries are from the "com" domain and Android libraries are well represented.

5 FUTURE RESEARCH QUESTIONS

This dataset opens avenues for several analyses. This section draws three research questions that this dataset could be used to address.

Table 3: Top 10 roots and fields present in AndroLibZoo

Top 10 roots		Top 10 most used fields		Top 10 most used fields w/o roots	
Root	**Count**	**Field**	**Count**	**Field**	**Count**
com	13 514	com	13 844	gradle	680
org	5450	org	5519	android	443
io	2630	io	2646	maven	352
net	1651	net	1704	com	330
de	1287	de	1288	plugins	269
cn	784	cn	784	sdk	267
dev	562	gradle	697	plugin	258
me	548	dev	575	co	244
eu	329	me	557	tools	164
se	304	co	460	api	153

Research Question 1: *How does AndroLibZoo compare with state-of-the- art approaches?* With this RQ, AndroLibZoo can be compared to existing techniques such as Li et al. [16] or Ma et al. [19]. Researchers can assess their comprehensiveness and precision.
Research Question 2: *To what extent can AndroLibZoo be useful and effective for static analysis?* This RQ would evaluate the importance of AndroLibZoo. Researchers can extract all packages from a dataset of apps and check the number of packages filtered using AndroLibZoo. This quantity can then be compared with the same approach using existing lists or more straightforward approaches, such as only considering the apps' package names to filter libraries.
Research Question 3: *Can AndroLibZoo improve the performance of existing static analysis tools?* With this RQ, AndroLibZoo can be used within existing static analyzers to check whether their performances, in terms of scalability and precision, is improved.

6 LIMITATIONS

One limitation of our work is that we did not consider all potential sources of third-party libraries publicly available. While we extracted libraries from Maven and Google's repositories, as well as open-source Android projects, there may be other sources of libraries. This limitation is mitigated by the fact that given our hypothesis, we believe that the sources of libraries we relied on are representative of how developers build apps in general.

Another limitation is that we only considered a subset of open-source Android projects when extracting libraries. While we used the AndroZoo and the F-Droid datasets as sources of open-source projects, there are likely additional open-source projects available.

Another limitation of our work is that our list of libraries is only designed to match non-obfuscated libraries. Our study has shown that this limitation is also mitigated (cf. Figure 2) since the majority of package names in Android apps are not obfuscated. Besides, the detection of obfuscated libraries is another research direction that is actively being explored by the literature [5, 25, 30].

7 CONCLUSION

In this paper, we presented an approach for automatically generating an accurate and up-to-date white list of third-party libraries that can serve the research and practitioner communities. Our dataset, AndroLibZoo, contains 34 813 package names which, by construction, only represent libraries, and is meant to evolve.

8 ACKNOWLEDGEMENT

This research was funded in whole, or in part, by the Luxembourg National Research Fund (FNR), grant references REPROCESS Project (16344458) and UNLOCK Project (18154263).

REFERENCES

[1] Aafer, Y., Du, W., and Yin, H. Droidapiminer: Mining api-level features for robust malware detection in android. In *Security and Privacy in Communication Networks* (Cham, 2013), T. Zia, A. Zomaya, V. Varadharajan, and M. Mao, Eds., Springer International Publishing, pp. 86–103.

[2] Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (New York, NY, USA, 2016), MSR '16, ACM, pp. 468–471.

[3] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN NOTICES 49*, 6 (June 2014), 259–269.

[4] Avdiienko, V., Kuznetsov, K., Gorla, A., Zeller, A., Arzt, S., Rasthofer, S., and Bodden, E. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (2015), ICSE '15, IEEE Press, p. 426–436.

[5] Backes, M., Bugiel, S., and Derr, E. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, Association for Computing Machinery, p. 356–367.

[6] Ban, Y., Lee, S., Song, D., Cho, H., and Yi, J. H. Fam: Featuring android malware for deep learning-based familial analysis. *IEEE Access 10* (2022), 20008–20018.

[7] Białecki, A., Muir, R., Ingersoll, G., and Imagination, L. Apache lucene 4. In *SIGIR 2012 workshop on open source information retrieval* (2012), p. 17.

[8] Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., and Yin, H. *Automatically Identifying Trigger-based Behavior in Malware.* Springer US, Boston, MA, 2008, pp. 65–88.

[9] Chen, K., Liu, P., and Zhang, Y. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, Association for Computing Machinery, p. 175–186.

[10] Fratantonio, Y., Bianchi, A., Robertson, W., Kirda, E., Kruegel, C., and Vigna, G. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE Symposium on Security and Privacy (SP)* (2016), pp. 377–396.

[11] Grace, M. C., Zhou, W., Jiang, X., and Sadeghi, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks* (New York, NY, USA, 2012), WISEC '12, Association for Computing Machinery, p. 101–112.

[12] Gradle. Google maven repositoryhttps://docs.gradle.org/current/userguide/declaring_repositories.html#sub:maven_google, 2022. Accessed December 2022.

[13] Gradle. Maven central repositoryhttps://docs.gradle.org/current/userguide/declaring_repositories.html#sub:maven_central, 2022. Accessed December 2022.

[14] JFrog. Jcenter, https://developer.android.com/studio/build/jcenter-migration, 2023. Accessed Apr. 2023.

[15] Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and McDaniel, P. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (2015), ICSE '15, IEEE Press, p. 280–291.

[16] Li, L., Bissyandé, T. F., Klein, J., and Le Traon, Y. An investigation into the use of common libraries in android apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2016), vol. 1, pp. 403–414.

[17] Li, L., Bissyandé, T. F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., and Traon, L. Static analysis of android apps: A systematic literature review. *Information and Software Technology 88* (2017), 67–95.

[18] Li, L., Riom, T., Bissyandé, T. F., Wang, H., Klein, J., and Yves, L. T. Revisiting the impact of common libraries for android-related investigations. *Journal of Systems and Software 154* (2019), 157–175.

[19] Ma, Z., Wang, H., Guo, Y., and Chen, X. Libradar: Fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion* (New York, NY, USA, 2016), ICSE '16, Association for Computing Machinery, p. 653–656.

[20] Maintainer, F. Flowdroid's systemclasshandler class https://github.com/secure-software-engineering/FlowDroid/blob/develop/soot-infoflow/src/soot/jimple/infoflow/util/SystemClassHandler.java, 2023. Accessed January 2023.

[21] Samhi, J., Bartel, A., Bissyande, T. F., and Klein, J. Raicc: Revealing atypical inter-component communication in android apps. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (Los Alamitos, CA, USA, May 2021), IEEE Computer Society, pp. 1398–1409.

[22] Samhi, J., Gao, J., Daoudi, N., Graux, P., Hoyez, H., Sun, X., Allix, K., Bissyandé, T. F., and Klein, J. Jucify: A step towards android code unification for enhanced static analysis. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (Los Alamitos, CA, USA, May 2022), IEEE Computer Society, pp. 1232–1244.

[23] Samhi, J., Li, L., Bissyande, T. F., and Klein, J. Difuzer: Uncovering suspicious hidden sensitive operations in android apps. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (Los Alamitos, CA, USA, May 2022), IEEE Computer Society, pp. 723–735.

[24] Sun, X., Chen, X., Li, L., Cai, H., Grundy, J., Samhi, J., Bissyandé, T. F., and Klein, J. Demystifying hidden sensitive operations in android apps. *ACM Trans. Softw. Eng. Methodol.* (dec 2022). Just Accepted.

[25] Wang, Y., Wu, H., Zhang, H., and Rountev, A. Orlis: Obfuscation-resilient library detection for android. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)* (2018), pp. 13–23.

[26] Wei, F., Roy, S., Ou, X., and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, Association for Computing Machinery, p. 1329–1341.

[27] Zhan, X., Fan, L., Liu, T., Chen, S., Li, L., Wang, H., Xu, Y., Luo, X., and Liu, Y. Automated third-party library detection for android applications: Are we there yet? In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2020), pp. 919–930.

[28] Zhan, X., Fan, L., Liu, T., Chen, S., Li, L., Wang, H., Xu, Y., Luo, X., and Liu, Y. Automated third-party library detection for android applications: Are we there yet? In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2021), ASE '20, Association for Computing Machinery, p. 919–930.

[29] Zhan, X., Liu, T., Fan, L., Li, L., Chen, S., Luo, X., and Liu, Y. Research on third-party libraries in android apps: A taxonomy and systematic literature review. *IEEE Transactions on Software Engineering 48*, 10 (2022), 4181–4213.

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