Revisiting Permission Piggybacking of Third-Party Libraries in Android Apps

Kris Heid[a], Elena Julia Sonntag and Jens Heider[b]
Fraunhofer SIT — ATHENE - National Research Center for Applied Cybersecurity,
Rheinstraße 75, 64295 Darmstadt, Germany
{*kris.heid, elena.julia.sonntag, jens.heider*}*@sit.fraunhofer.de*

Keywords: Permission, Third-Party Libraries, Android, Static Analysis.

Abstract: Permissions have been employed to let the user decide on components an app can interact with. However, apps typically consist of the main app along with several libraries to support the developer with various functionality and tasks. The fact that libraries inherit the permissions of the main app gives these libraries often more rights than needed for their core functionality. Many libraries do permission piggybacking and thus probe available permissions without requesting permissions themselves and adapt their behavior accordingly. Especially, advertisement and tracking libraries show high interest to collect as much user data as possible through this technique. Many works have previously addressed this problem but no solution has made its way into Android. This work delivers a novel analysis technique agnostic to the Android API level without manual mapping effort like previous works. Our results show, that permission piggybacking remains a problem to be urgently addressed.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

There exists a gigantic number of apps in the Google Play Store for Android users to install on their phones. They are easy to install and use, thus many services offer an own app to the users. With so many apps and some of relatively unknown developers, the users demand control over which information and functionality an app is allowed to use. Thus, permissions came up relatively early in the Android development to inform the user about capabilities of the app before installation. Later on, this system was refined with runtime-permissions to let the user decide to provide access to resources during runtime of the app. The user must thus try to make an informed decision during the app's runtime if access to certain functionality is appropriate or not, which is often difficult.

However, apps internally often use third-party libraries to offload some programming effort. With the use of libraries, app development becomes faster and easier but even app developers don't know the exact internals of the third-party libraries they use. Third-party libraries can have background activities running besides their advertised functionality. These libraries then have the same access to functionality and information as the main app since they run under the same process in the OS. Thus, the permission management system cannot distinguish the main app from libraries(Stevens et al., 2012; Zhao et al., 2023). There already exist numerous publications since circa 2015 (see section 2) on how to implement permissions on finer granularity level and have different permissions for the main app and third-party libraries to mitigate this issue. However, ever since none of such solutions made it into the Android source-code. A possible explanation might be that additional permission requests would be a big burden to the user for the benefits it brings.

Before starting our research, we were already aware of advertisement and tracking libraries to use a technique one could call opportunistic permission usage or permission piggybacking. With this technique third-party libraries don't request permissions themselves, but use what is already granted to the main app. As a result, such libraries get access to much information and critical functionality when used in an app with high privileges.

In this publication, we want to find out the extent of opportunistic permission usage by third-party libraries and thus if it is a serious problem which must be addressed or a niche which would justify Google's

[a] https://orcid.org/0000-0001-7739-224X
[b] https://orcid.org/0000-0001-8343-6608

Heid, K., Sonntag, E. J. and Heider, J.
Revisiting Permission Piggybacking of Third-Party Libraries in Android Apps.
DOI: 10.5220/0013126500003899
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 11th International Conference on Information Systems Security and Privacy (ICISSP 2025) - Volume 1*, pages 39-46
ISBN: 978-989-758-735-1; ISSN: 2184-4356
Proceedings Copyright © 2025 by SCITEPRESS – Science and Technology Publications, Lda.

reserve to implement proposed solutions into Android.

The remainder of this paper is structured as following: In section 2 we will address related work about this topic and what are the contributions of this publication. Section 3 describes the concepts and implementation details how we detect opportunistic permission usage of third-party libraries. In section 5, we analyze the top 1,000 apps on Google Play with our detector and evaluate the results. The last section gives a conclusion and outlook on future work.

2 RELATED WORK

We divide this section into three parts. In the first part, we present works which analyze permissions of third-party libraries. The second part presents concepts to separate the permissions of the main app and the libraries for fine granular control over the libraries abilities. In the third part, we highlight the planned contributions of this work in contrast to former publications.

2.1 Permission Analysis

Stevens et al. (Stevens et al., 2012) compared in-browser and in-app advertising regarding user privacy and found that in-app advertising is more likely to compromise user privacy. They examined 13 popular Android advertising libraries to determine the risk of privacy violations concerning permissions, categorizing them into required, optional, and undocumented permissions. These undocumented permissions are dynamically checked and used by libraries, allowing access to additional sensitive data if the application has these permissions. Using the Stowaway tool (Felt et al., 2011), they discovered additional undocumented permissions and confirmed their actual usage via manual checks, using a hybrid analysis approach (Zhan et al., 2021).

Book et al. (Book et al., 2013) studied the evolution of advertising libraries in Android concerning permissions and their impact on privacy and security. They used PScout (Au et al., 2012) to analyze system calls requiring permissions and found that advertising libraries increasingly exploit permissions requested by the main app. This concept was further refined by Backes et al. (Backes et al., 2016) to cover new Android versions and improve accuracy.

Grace et al. (Grace et al., 2012) also investigated privacy and security risks from embedded advertising libraries in Android apps. Analyzing 100,000 apps, they identified 100 advertising libraries and used the AdRisk tool to identify potential risks. Unlike (Book et al., 2013), this study includes Android API calls not requiring permissions and identifies ”Permission Probing”, where advertising libraries opportunistically use APIs requiring permissions, either by checking permissions in advance or handling `SecurityException`. More than half of the examined libraries exhibit this behavior.

2.2 Third-Party Library Permission Isolation

Bin Liu et al. (Liu et al., 2015) introduced PEDAL to de-escalate the privileges of advertising libraries, with two components: the Separator and the Rewriter. The Separator identifies advertising libraries by analyzing Java bytecode and separates them from app logic. The Rewriter controls resource access by advertising libraries and prevents permission inheritance. They identified features for classifying advertising libraries, including APIs for permission checking like `android.content.Context.check*Permission()`. Analysis revealed that five of 30 libraries use undocumented permissions (Zhan et al., 2021).

Several publications present tools to isolate third-party libraries from the host app, with two approaches: separating libraries into independent processes (1) and interrupting communication between libraries and host apps (2) (Zhan et al., 2021). Tools like AdSplit (Shekhar et al., 2012), AdDroid (Pearce et al., 2012), SanAdBox (Kawabata et al., 2013), AFrame (Zhang et al., 2013), NativeGuard (Sun and Tan, 2014), and FLEXDROID (Seo et al., 2016) follow approach (1), while PEDAL(Liu et al., 2015) and LibCage (Wang et al., 2016) follow approach (2). For completeness, recent research by Zhan et al. (Zhan et al., 2021) provides a comprehensive overview.

2.3 Contribution

This work does not aim to show another third-party library permission isolation approach. Instead, we want to analyze how widespread the opportunistic permission usage is in all third-party libraries, not limited to advertisement libraries as many previous publications. Previous works often use a complex ”required permission” to ”restricted API-call” mapping (Stevens et al., 2012; Felt et al., 2011; Book et al., 2013; Grace et al., 2012) optionally with call graph reachability analysis(Backes et al., 2016) to find out where permissions are requested and used. This, approach becomes outdated each year with new Android versions and therewith new permissions and APIs and is thus cumbersome to manually go through

API changes and keep the mapping up to date especially considering that Backes et al.(Backes et al., 2016) mentioned many undocumented APIs.

In contrast, our approach tries to find `checkPermission(...)` and `requestPermission(...)` pairs inside libraries and match them. Normally, a library would check the same permissions as it requests. The price to pay of not having API-call to permission mapping is missing out on permission probes which make restricted API calls and catch the security exception. The advantage of this method is its remaining compatibility with future Android releases. An additional contribution is, that we not only focus on advertisement libraries like previous publications, but any library. We prove the applicability of our approach on the top 1,000 apps on Google Play.

3 CONCEPT AND GOAL

To classify libraries as normal or opportunistic permission usage, the `checkPermission()` and `requestPermission()` calls must be compared with the permissions passed to these calls.

For an APK, all its third-party libraries must be identified. All classes of this library are examined for `checkPermission()` and `requestPermissions()` calls. If a `checkPermission()` call is found, the checked permission is added to a list for this library. The same procedure is applied for `requestPermissions()` calls and all requested permissions are added to a list for this library. A library with opportunistic permission usage would have more items in the check permissions list than in the request permissions list. All methods and classes to check and request permissions are shown in Table 2.

The following example in Table 1 should be an APK that has integrated two third-party libraries `com.library1` and `com.library2`.

Table 1: Library Permissions.

	permission	
Library	checked	requested
com.library1	RECORD_AUDIO	RECORD_AUDIO
com.library2	FINE_LOCATION RECORD_AUDIO	RECORD_AUDIO

`com.library1` would be “normal” privileged, as both lists for requested and checked permissions contain the same elements (permissions). However, `com.library2` would be count as permission piggybacking, as the checked permissions contains the `FINE_LOCATION` permission, which is not found in the requested permissions. The library would thus probably like to piggyback on the main app’s `FINE_LOCATION` permission.

Table 2: API Methods to check or request permissions.

Method	Class
checkPermission()	Context PackageManager PermissionChecker
checkSelfPermission()	ContextCompat ActivityCompat Context PermissionChecker
checkCallingOrSelfPermission()	Context PermissionChecker
requestPermissions()	ActivityCompat Activity

3.1 Resolve Arguments

So far, it has been assumed that constants in the form of strings are passed to the permission function calls, which is not necessarily the case. Sometimes permissions are assigned to variables beforehand, which are then passed to the function calls. That is, calls like `checkSelfPermission(context, permissions)` or `requestPermissions(context, permissions, requestCode)`. However, to compare specific, readable permissions, these variables must be resolved. In easy cases the call site of the function call to for example `checkPermission(...)` needs to be found. In complex cases the data flow has to be chased through various function calls and variable assignments or even through Java reflections. Since not all data flows are resolvable in a static analysis, we add a list of unresolvable variables for each checked and requested permissions. In section 4.6 we will go into further detail how libraries can be identified in some cases as permission piggybacking even though containing unresolved variables. At this point unresolvable arguments might seem as a big drawback, but as later shown in the evaluation section 5, the situation rarely happens in practice.

4 IMPLEMENTATION

In this section we will explain the details of our implementation. Before starting with an implementation we researched for fitting tools to get the job done.

4.1 Tool Selection

Due to several previous works, there should presumably be some analysis tools available to use. Our requirements for the tool are:

1. Handle Android APK files
2. Ability to search in Java classes for specific function calls
3. Call graph analysis abilities to resolve function call arguments (permissions)
4. Easy to use and install

From our literature research in section 2 it would be easiest to take an existing tool and slightly adopt it for our use case. However, it was barely possible: Stowaway (Stevens et al., 2012) source-code was no longer available on the internet and they direct users towards using PScout. PScout in turn remains unmaintained since its initial commit 9 years ago, thus probably no good choice either. The slightly later released Axplorer (Backes et al., 2016) has a github[1] with permission to API call mappings, but unfortunately no source-code. The source-code from other tools was not published like for (Grace et al., 2012) or the adDetect tool (Narayanan et al., 2014). However, most of these tools also relied on common Java or Android APK analysis frameworks such as WALA[2], soot[3], APKtool[4] or randoop[5]. Randoop is a Java unit test generator used by Grace et al.(Grace et al., 2012) as a search engine for function calls. However, this tool doesn't offer fancy analysis functionality like call graph analysis and many more. Additionally, tools like APKtool need to be used to decompile the app. APKtool used by Narayanan et al.(Narayanan et al., 2014) in turn only decompiles without any code analysis functionality. Better suited seems WALA used by Backes et al.(Backes et al., 2016), which seems to have all source-code analysis functionality we need. After looking at the documentation, we saw that even for the simple "getting started" examples, several hundreds lines of code are required, which seems like a big overhead. Additional steps like APKtool would also be necessary to analyze APKs. Soot is used internally by many tools like Stowaway or PScout, and it would also provide required code analysis functionality. After looking through the code we realized the frequent use of singletons which prevents parallelization, which we would want when analyzing 1,000 apps. JADX[6] is also a very popular APK analysis tool including a decompiler. Recently, it also separated its core functionality from the GUI which now makes it a good tool for programmatically interact with APK files. Interestingly, it wasn't used yet in related work. With its ease of used all our requirements are met and is thus the tool for our analysis.

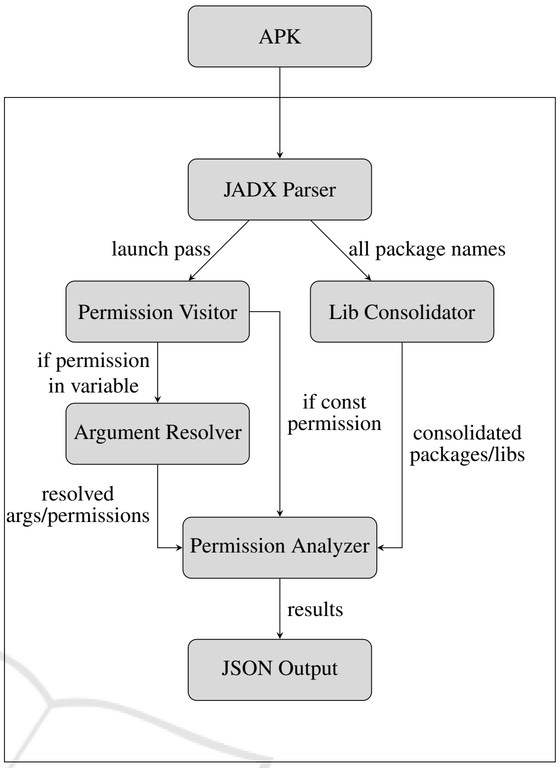

Figure 1: Analysis tool flow chart.

[1]https://github.com/reddr/axplorer

[2]https://github.com/wala/WALA

[3]https://github.com/soot-oss/soot

[4]https://apktool.org/

[5]https://github.com/randoop/randoop

[6]https://github.com/skylot/jadx

4.2 Implementation Details

In this chapter, the structure and implementation of our analysis tool are described. First, a brief explanation of the general flow of a program run is given. Then, the most important components of the program are discussed in detail.

At the beginning, an Android app (APK) is fed into the tool. The JADX parser starts analyzing and decompiling the APK to reflect all components in its internal representation. The *Permission Visior* is our own analysis pass which runs after several JADX integrated passes have finished. The *Permission Visitor* searches all classes for calls to `checkPermission(...)` and `requestPermission(...)` function calls. For each found call, the permission and the class package it was found in is passed to the *Permission Analyzer*. However, if the argument of the permission request/check function call is not directly a constant, the argument has to be resolved through the *Argument Resolver*. The *Argument Resolver* traces back possible data flows and tries to find the definition of the argument. This process is well-supported through the JADX representation. The *Lib Consolidator* runs in parallel to the previous steps. It receives a list of all package names of the contained classes in the APK and tries to merge them together. For example, if the provided package names are `com.main.app`, `org.lib.asd`, `org.lib.fu` the packages are consolidated to `com.main.app`, `org.lib`. Afterward, the *Permission Analyzer* has a list of requested and checked permissions per package. Through the consolidation, the packages are grouped to library names and the permission piggybacking analysis can start. In the easiest case, the requested permissions equal the checked permissions. If permission piggybacking occurs, the request permission list don't match the check permission list of a library package. More difficult cases arise of permission names that cannot be resolved through the *Argument Resolver* which will be explained in detail in section 4.6. When the analysis finishes, a JSON output is created with the analysis results, containing libraries with permission piggybacking and which permissions are piggybacked.

4.3 Permission Visitor

The *Permission Visitor* is responsible for going through all the methods of a class that is currently being decompiled and identify and filter out the permission calls `checkPermission(...)` and `requestPermissions(...)` within these methods. Before the permissions are handed over to the *Permission Analyzer*, the *Permission Visitor* check whether the permissions in the respective function calls are passed as constants or variables. If a variable is passed to the permission functions, the *Argument Resolver* is called to tries to resolve this variable. If constants are passed, they can be directly collected and passed to the *Permission Analyzer*.

The `Permission Visitor` works closely with the *JADX Parser* to enable the decompilation and analysis of APK files. The *JADX Parser* initializes and loads the APK file, prepares the necessary environment, and adds the *Permission Visitor* as a custom Dex tree visitor. The `Permission Visitor` is added as a custom pass to the list of Dex tree visitors (*passes*) of the *JadxDecompiler*. When the Dex tree of the APK is traversed, the *JadxDecompiler* calls the `visit` method of the *Visitor* object on each method in the APK, which performs the check for a permission call.

4.4 Argument Resolver

The task of the *Argument Resolver* is to resolve the value of arguments which are not constants. Mostly, permission requests directly use the provided permission constants. However, one could also define a variable and assign the constant or the variable could be passed through the arguments of a wrapper function and many more scenarios are thinkable. As possibilities are manifold. We use a pragmatic approach and target to resolve 90% of the real-world cases and ditch corner cases. Such corner cases like calls through reflection, indexed array access or loops are extremely difficult or yet impossible to properly resolve in our chosen static context. For example, symbolic execution is one research topic to resolve such variables. Since such techniques are not the focus of this paper we allow ourselves to omit such cases.

Our implemented resolver can thus resolve local variables, instance variables, class variables as well as invoked functions to return the permission(s) and permissions passed through the arguments of a wrapper function. The *Argument Resolver* uses recursive resolving and therewith trace back the permissions. For example, if a permission is assigned from a class variable to a local variable and then to a `checkPermission(...)` function call, the *Argument Resolver* is called twice recursively first to resolve the local variable and a second time to resolve the class variable. It is also possible to have multiple data sources which could resolve the permission arguments.

After running our resolver on the top 1,000 apps on Google Play, we have evaluated that we can resolve 90% of the permissions passed to respective permission functions. These results are considered good enough for the purpose of showing the feasibility of our approach but could be further improved in the margins of what is possible in a static context.

4.5 Lib Consolidator

Library consolidation is necessary, since our permission analysis search only returns in which classes request or check permission calls are found. These calls can be spread over multiple classes but are dedicated to the same library. For example, class `com/lib/sub1/one.class` and `com/lib/sub1/one.class` in fig. 2 are located in different packages (com/lib/sub1 and com/lib/sub2) but are part of the same library (com/lib). Through observation of naming conventions, we were able to constitute a few heuristics which we use to consolidate library names:

```
APK
 ├─com/lib/
 │  ├─sub1/
 │  │  └─One.class
 │  ├─sub2/
 │  │  └─Other.class
 │  └─Main.class
 └─org/other
    └─MyMain.class
```

Figure 2: Folder Structure and package hierarchy in APKs.

- Library names are at least two levels deep (com.library)
- Ascending directories, the folder with the first class file defines the library name. All class files in lower directories are part of this library.

With this heuristic, we would extract the two libraries: `com.lib` and `org.other` from an example APK shown in fig. 2. When obfuscation is applied, typically package flattening is done. Thus, prior rules are not applicable any more and a library consolidation not useful and therewith skipped.

4.6 Permission Analyzer

The *Permission Analyzer* takes the results from the previous steps and consolidates the permissions to the grouped library names. For each library in the app, there exist four lists: checked permissions, requested permissions, unresolved checked permissions and unresolved requested permissions. If no unresolved permissions exist, the check is simple and can be achieved by checking if the set of checked permissions is contained in the set of requested permissions. If this holds true, **no** permission piggybacking is found and if it is false, permission piggybacking is found. However, since unresolvable permissions are possible, we set up a truth table in table 3 on how to proceed in every possible scenario. A "1" in the table marks where the respective list contains entries and the "0" indicates an empty list. Most cases are self-explanatory, only rows 14 and 15 might require more explanation:

Row 14: In this step, first checked permissions and requested permission lists are checked for equivalence. If they are equal, one can safely say that no permission piggybacking appears. If they are not equal, the unresolved permission might contain the necessary permission, but one cannot surely say, and thus we would justify it as *unknown*.

Row 15: Again we first compare checked permissions and requested permission lists for equivalence. If they are not equal, permission piggybacking does happen. If they are equal, the unresolvable checked permissions might already be contained in the checked permission list or not, and thus the result would be *unknown*

Table 3: Truth table to judge for piggybacking with unresolvable permissions. ((U)chk=(unresolvable) check permission, (U)req=(unresolvable) request permission).

	chk	req	Uchk	Ureq	perm. piggybacking
1	0	0	0	0	no
2	0	0	0	1	no
3	0	0	1	0	yes
4	0	0	1	1	unknown
5	0	1	0	0	no
6	0	1	0	1	no
7	0	1	1	0	unknown
8	0	1	1	1	unknown
9	1	0	0	0	yes
10	1	0	0	1	unknown
11	1	0	1	0	yes
12	1	0	1	1	unknown
13	1	1	0	0	regular check
14	1	1	0	1	no or unknown
15	1	1	1	0	yes or unknown
16	1	1	1	1	unknown

5 EVALUATION

For the evaluation, we used the top 1,000 apps[7] from Google Play and run our analysis on them to see how widespread permission piggybacking is. Out of all processed apps, a total of 851 different libraries were found. In fig. 3 50% of the libraries exhibited permission piggybacking, while 36% of the libraries either did not check for permissions or also requested permissions which were checked. 14% of the libraries were not safely assessable due to unresolvable permissions. These numbers are similar to the ones found by Grace et al.(Grace et al., 2012) in 2012. Which means that seemingly Google didn't find it necessary to tackle this problem. Considering that, as noted in the introduction, libraries make up about 60% of the app code, this observation is quite alarming. Although checking for permissions without a subsequent pop-up does not necessarily mean that this library exhibits privacy-threatening behavior, it has the potential to do so. For example, it could spy on, collect user data, and send it to other third parties or external servers.

The permissions which are most often checked without also requesting permission are the fine and coarse location as well as read phone state. From

[7]https://42matters.com/top-charts-explorer

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

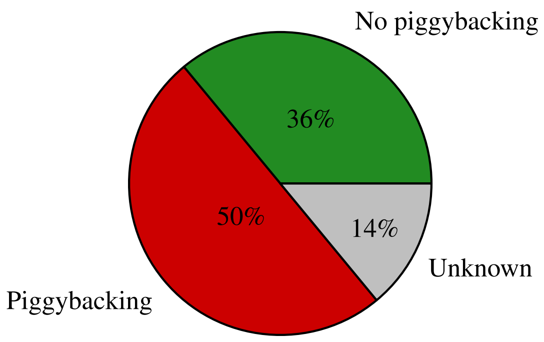

Figure 3: Analysis results for libraries piggybacking permissions.

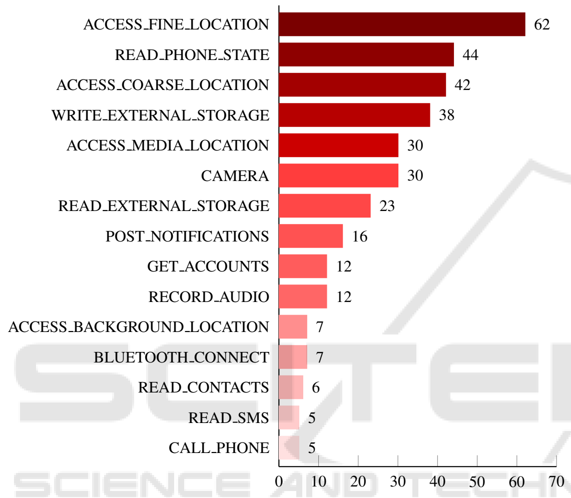

Figure 4: Popular permissions while piggybacking.

a privacy perspective, these results are quite alarming. Especially the location permission allows libraries for example to track the user location. Also, the *READ_PHONE_STATE* permission allows the library to collect some identifiers which are quite popular for device fingerprinting(Heid and Heider, 2024). That means that with the three most piggybacked permission a library can uniquely identify a device and therewith a user and also track the movement. Looking at the 15 most popular piggybacked permissions, it becomes obvious that 10 out of these 15 are declared *dangerous* permissions in Android terminology.

In the next step we analyzed most popular libraries doing permission piggybacking. We don't want to call out any names for legal reasons, but the top ten list exclusively contained libraries known for advertising, tracking and statistics. This seamlessly blends into the types of the checked permissions of the previous part. From Heid et al. (Heid and Heider, 2024) we know that such libraries typically transmit a lot personal identifiable information available through such permissions to their servers. A finer granular permission system on library level would solve such issues and is desirable from what we've seen.

The only library on the top list but not fitting into the picture is `androidx.appcompat`. This library is part of the Android Jetpack libraries and provides a set of support libraries that allow developers to access new APIs in older platform API versions and help create modern Android apps that are compatible with older Android versions[8].

5.1 Comparing Numbers from Related Work

By comparing our results with previous related work we are on the one hand able to reflect the evolution of Android permissions and also the evolution of apps.

Even though Stevens et al.(Stevens et al., 2012) and Grace et al. (Grace et al., 2012) have solely concentrated on advertisement apps, the permission types which are commonly piggybacked remain even after 12 years the same. Both works list the location and reading phone status as the most piggybacked permissions just like we discovered. Interestingly, Grace et al. say that seldom dangerous APIs in respect to the required dangerous permission are used. In contrast, we identified that mostly dangerous permissions were piggybacked. This might be since throughout the last 10 Android versions many new permissions were added, some were deprecated, and others were split into more granular permissions increasing the number of dangerous permissions.

Comparing the performance of our tool with Grace et al. we can process the top 1,000 apps at an average rate of 6 seconds per app on a 13th gen i7 processor, while their tool took 15 seconds per library. However, these numbers are not comparable due to a 12 year newer processor. Also, their tool works on already decompiled libraries, while the majority of our time goes into app decompilation.

6 CONCLUSION & FUTURE WORK

Six to twelve years old publications have already shown that there is a huge gap in protecting user privacy and security with Androids current permission system. Works have shown suggestions to resolve this, but until today no such concepts found their way

[8]https://developer.android.com/jetpack/androidx

to mainstream Android. In this work, we revisited the extent of permission piggybacking in libraries. Our proposed method works on any Android version and is, in contrast to other work, not reliant on the tedious manual API extraction and mapping to permissions. However, we have shown that our approach delivers identical evaluation result compared to previous publications. In our evaluation we were able to analyze all libraries on the top 1,000 apps on Google Play and did not limit our search to advertisement libraries. Even though, most piggybacking libraries we discovered fell into advertisement and tracking category. Most popular piggybacked permissions remain almost identical to the ones prevalent 12 years ago. Also, the most alarming number that 50% of all libraries use permission piggybacking remain constant throughout the years.

To change this practice, this topic needs more attention in the media to become relevant for Google to implement means into Android. What changed in the past 12 years is that privacy has gained more attention among normal users of digital platforms. The previous Android releases had strong privacy focus, like for example Scoped Storage in Android 11 or the Privacy Dashboard in Android 12. Thus, if permission piggybacking comes into focus again, maybe it would draw Googles attention with their recent privacy initiatives. Another option would be to implement a permission piggybacking prevention system in privacy focused custom ROMs such as GrapheneOS. It wouldn't be the first feature finding its way from a custom ROM into the official Android version.

ACKNOWLEDGEMENTS

This research work was supported by the National Research Center for Applied Cybersecurity ATHENE and the Hessian Ministry of the Interior and Sports.

REFERENCES

Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. (2012). Pscout: analyzing the android permission specification. In *ACM CCS*, pages 217–228.

Backes, M., Bugiel, S., Derr, E., McDaniel, P., Octeau, D., and Weisgerber, S. (2016). On demystifying the android application framework: Re-Visiting android permission specification analysis. In *USENIX Security*, pages 1101–1118, Austin, TX.

Book, T., Pridgen, A., and Wallach, D. S. (2013). Longitudinal analysis of android ad library permissions. *arXiv:1303.0857*.

Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. (2011). Android permissions demystified. In *ACM CCS*, pages 627–638.

Grace, M. C., Zhou, W., Jiang, X., and Sadeghi, A.-R. (2012). Unsafe exposure analysis of mobile in-app advertisements. In *ACM WiSec*, pages 101–112.

Heid, K. and Heider, J. (2024). Haven't we met before? - detecting device fingerprinting activity on android apps. EICC '24, page 11–18. ACM.

Kawabata, H., Isohara, T., Takemori, K., Kubota, A., Kani, J., Agematsu, H., and Nishigaki, M. (2013). Sanadbox: Sandboxing third party advertising libraries in a mobile application. In *ICC*, pages 2150–2154. IEEE.

Liu, B., Liu, B., Jin, H., and Govindan, R. (2015). Efficient privilege de-escalation for ad libraries in mobile apps. In *MobiSys*, pages 89–103.

Narayanan, A., Chen, L., and Chan, C. K. (2014). Addetect: Automated detection of android ad libraries using semantic analysis. In *IEEE ISSNIP*, pages 1–6.

Pearce, P., Felt, A. P., Nunez, G., and Wagner, D. (2012). Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 71–72.

Seo, J., Kim, D., Cho, D., Shin, I., and Kim, T. (2016). Flexdroid: Enforcing in-app privilege separation in android. In *NDSS*.

Shekhar, S., Dietz, M., and Wallach, D. S. (2012). {AdSplit}: Separating smartphone advertising from applications. In *USENIX Security*, pages 553–567.

Stevens, R., Gibler, C., Crussell, J., Erickson, J., and Chen, H. (2012). Investigating user privacy in android ad libraries. In *Workshop on Mobile Security Technologies (MoST)*, volume 10, pages 195–197.

Sun, M. and Tan, G. (2014). Nativeguard: Protecting android applications from third-party native libraries. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 165–176.

Wang, F., Zhang, Y., Wang, K., Liu, P., and Wang, W. (2016). Stay in your cage! a sound sandbox for third-party libraries on android. In *Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I 21*, pages 458–476. Springer.

Zhan, X., Liu, T., Fan, L., Li, L., Chen, S., Luo, X., and Liu, Y. (2021). Research on third-party libraries in android apps: A taxonomy and systematic literature review. *IEEE Transactions on Software Engineering*, 48(10):4181–4213.

Zhang, X., Ahlawat, A., and Du, W. (2013). Aframe: Isolating advertisements from mobile applications in android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 9–18.

Zhao, K., Zhan, X., Yu, L., Zhou, S., Zhou, H., Luo, X., Wang, H., and Liu, Y. (2023). Demystifying privacy policy of third-party libraries in mobile apps. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1583–1595. IEEE.