AutoDebloater: Automated Android App Debloating

Jiakun Liu*, Xing Hu†, Ferdian Thung*, Shahar Maoz‡, Eran Toch§, Debin Gao*, and David Lo*
*School of Computing and Information Systems, Singapore Management University, Singapore
†School of Software Technology, Zhejiang University, China
‡School of Computer Science, Tel Aviv University, Israel
§Department of Industrial Engineering, Tel Aviv University, Israel
{jkliu, ferdianthung, dbgao, davidlo}@smu.edu.sg, xinghu@zju.edu.cn, maoz@cs.tau.ac.il, erant@tauex.tau.ac.il

***Abstract*—Android applications are getting bigger with an increasing number of features. However, not all the features are needed by a specific user. The unnecessary features can increase the attack surface and cost additional resources (e.g., storage and memory). Therefore, it is important to remove unnecessary features from Android applications. However, it is difficult for the end users to fully explore the apps to identify the unnecessary features, and there is no off-the-shelf tool available to assist users to debloat the apps by themselves. In this work, we propose AutoDebloater to debloat Android applications automatically for end users. AutoDebloater is a web application that can be accessed by end-users through a web browser. In particular, AutoDebloater can automatically explore an app and identify the transitions between activities. Then, AutoDebloater will present the Activity Transition Graph to users and ask them to select the activities they do not want to keep. Finally, AutoDebloater will remove the activities that are selected by users from the app. We conducted a user study on five Android apps downloaded from three categories (i.e., Finance, Tools, and Navigation) in Google Play and F-Droid. The results show that users are satisfied with AutoDebloater in terms of the stability of the debloated apps and the ability of AutoDebloater to identify features that are never noticed before. The tool is available at http://autodebloater.club. The code is available at https://github.com/jiakun-liu/autodebloater/ and the demonstration video can be found at https://youtu.be/Gmz0-p2n9D4.**

***Index Terms*—Android, Software Debloating**

I. Introduction

Nowadays, Android phones are becoming increasingly popular [1]. A large number of Android applications (i.e., apps) are shared in the app stores [2]. To meet the diverse needs of different users, Android apps often integrate a series of different functions into one, becoming a super app. For example, the Instagram app not only allows users to share and browse photos and videos but also provide "reels" in a dedicated tab.[1] The existence of these functions is driven by the app providers' desire to increase service sales and attract a larger user base. However, in many apps, many of these functions are not needed by the users, resulting in a bloated app [3]. Prior studies showed that 80% of features in average software products are rarely or never used [3]. This practice disregards the users' experience. Additionally, a large amount of code unrelated to the users runs on Android phones, resulting in wasted resources (such as battery power and memory usage) and potentially introducing potential risks [4]. Therefore, the bloated features in Android apps have become a problem.

[1]https://play.google.com/store/apps/details?id=com.instagram.android

Figure 1 shows an example of the Activity Transition Graph (ATG). ATG demonstrates which activity can be accessed from another. An activity is an application component that provides a screen with which users can interact to do something.[2,3] The graph in Figure 1 specifies 19 out of the total 35 activities in the Wikipedia app. For example, users can use Wikipedia to read articles, search for information, edit articles, read the saved articles, as well as create accounts, log in, log out, and reset passwords. However, users may not need all these features. For example, a user may just want to read articles and not want to log in or reset passwords to edit articles. For such users, the features related to account management and editing are redundant.

Prior studies proposed several solutions to debloat Android apps. For example, Jiang et al. considered the dead code as bloated code and identified the dead code in Android apps statically [5]. Pilgun considered the code that will never be executed as bloated code. He identified bloated code by collecting code coverage information through automated app exploration (i.e., a fuzzer). [6]. Then, He removed the bloated Smali code from the app and recompiled the app. Tang et al. identified features from different perspectives (e.g., permission, activity, and modularity) statically, and ask developers to select the features they want to keep [7].

However, these solutions failed to identify the features that are undesired by users. For users, debloating an app presents a challenge due to several reasons. Firstly, fully exploring an app and identifying unnecessary features can be difficult. Users may not be aware of the existence of certain features or functionalities that are unnecessary for their specific needs, i.e., they are unaware of the bloated features in the app. If so, they cannot request the removal of these features. For example, Figure 1 shows that there are 35 activities in the Wikipedia app, and the ATG is very complex. If users do not conduct a thorough exploration, they will not discover these bloated activities in the app. However, expecting users to fully explore an app is highly challenging [8]. Besides, there is no off-the-shelf tool available to assist them in debloating the app. Existing solutions require setting up a complete Android software development environment [7]. This is impractical for end-users of Android phones because not all users with app

[2]https://developer.android.com/guide/components/activities/intro-activities
[3]https://developer.android.com/reference/android/app/Activity

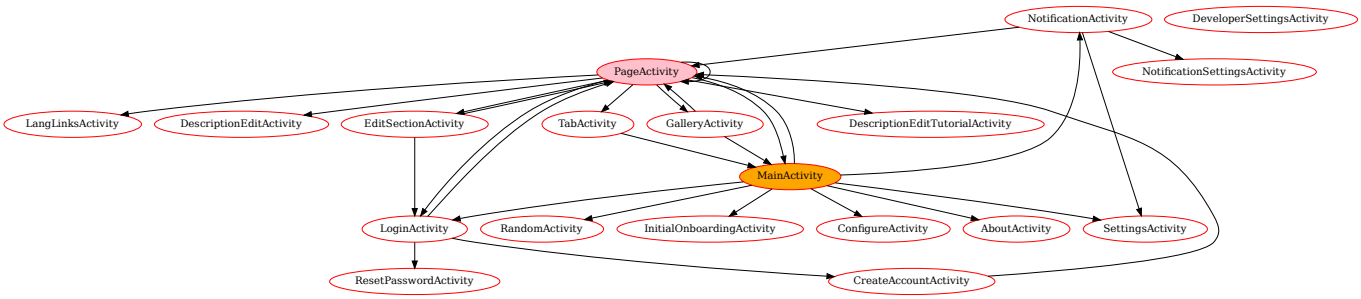

Fig. 1. An example of a part of the Activity Transition Graph of the Wikipedia app. This example shows that there is a large number of features in an app, and it can be difficult for a user to fully explore the app.

debloating needs are software developers. Therefore, a tool is needed to help end-users debloat Android apps.

To fill the gap, we propose AutoDebloater, a tool that can automatically debloat Android apps. AutoDebloater is a web application that can be accessed by end-users through a web browser. In particular, AutoDebloater can automatically explore an app and identify the transitions between activities using StoryDistiller [8], [9]. StoryDistiller is a state-of-the-art tool that can automatically explore Android apps and generate the ATG. Then, AutoDebloater will present the ATG to users and ask them to select the activities they want to keep. Finally, AutoDebloater will remove the activities that are not selected by users from the app.

To evaluate the performance of AutoDebloater, we conducted a user study on five Android apps downloaded from three categories (i.e., Finance, Tools, and Navigation) in Google Play and F-Droid. We asked seven users to use AutoDebloater to debloat the apps. Results show that users are satisfied with the stability of the debloated apps generated by AutoDebloater as well as the ability of AutoDebloater to help them identify the activities that they never noticed before. We also collect the time to debloat the apps using AutoDebloater. The results show that AutoDebloater can debloat apps in 20 seconds on average.

II. AutoDebloater

In this section, we first present the design of the website of AutoDebloater by providing a usage scenario. Then we introduce the background technique of AutoDebloater, including how we use StoryDistiller to extract the ATG, and how we remove the activities from the app.

A. Usage Scenario

Consider a user Alice, who just wants to browse the knowledge shared on Wikipedia, and does not want to edit the articles. She wants to remove the features related to editing articles from the Wikipedia app. Alice can use our tools to debloat the Wikipedia app. First, Alice needs to upload the Wikipedia APK to our server. Our server will analyze the APK using StoryDistiller to (1) identify all activities in the app, (2) take a screenshot of each activity, and (3) generate the ATG. After finishing the analysis, our server will send back the ATG as well as the list of all activities of the app to Alice.

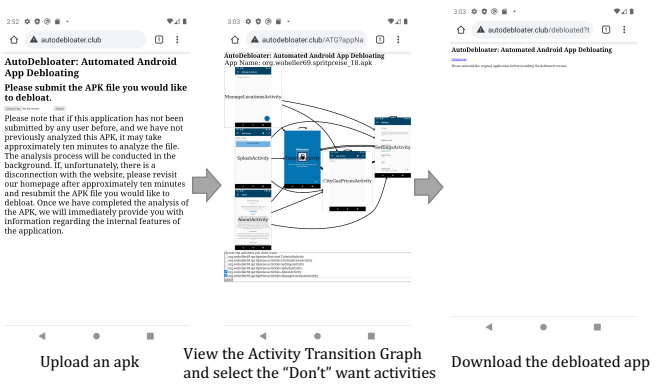

Fig. 2. An example of the use of AutoDebloater.

If the same APK has been uploaded before, our server will directly return the ATG to Alice without re-analyzing the APK, which can reduce the response time. Then, Alice can select the activities that she wants to keep. Our server will remove the activities that Alice does not want to keep from the app. After that, our server will recompile the app and send the debloated APK to Alice. Finally, Alice can install the debloated APK with all the features that Alice wants.

B. Extracting ATG using StoryDistiller

StoryDistiller is a state-of-the-art tool that can automatically explore Android apps and generate the ATG [8], [9]. Given an app to analyze, StoryDistiller outputs the (1) ATG of the app, together with (2) the screenshots of each activity in the app. The details of StoryDistiller can be found in the original papers [8], [9]. Still, to make our paper self-contained, we briefly introduce the working principle of StoryDistiller in this section.

To explore the transition from one activity to another, StoryDistiller needs to launch the app activities externally (outside the app, i.e., from the command line). Whether an activity can be launched externally is set by $android : exported$ in the $AndroidManifest.xml$ file. However, the default value of $android : exported$ is $false$, indicating that the activity cannot be launched by components of other apps. To solve this problem, StoryDistiller instruments the app and modifies the $AndroidManifest.xml$ file of the app by setting $android : exported$ to $true$ for all activities. By doing so,

StoryDistiller can launch the activities externally and take screenshots of each activity in the app.

Following that, StoryDistiller performs static program analysis to generate the static ATG. More specifically, StoryDistiller takes as input an app, then generates its call graph (CG). Then, StoryDistiller traverses each method of each class to obtain the explicit activity transition. More specifically, StoryDistiller first analyzes the intent constructor created in the method body and then tracks the parameter that indicates the target activity by data-flow analysis. If the method is in a Fragment (i.e., a reusable portion hosted in an activity)[4], StoryDistiller also identifies the activities corresponding to the fragment and merges fragment relations to construct the actual activity transitions.

During the static extraction process, StoryDistiller also extracts the Inter-Component Communication (ICC) data to collect information (i.e., primitive attributes and extra parameters) to launch the activities externally. More specifically, StoryDistiller parses the manifest file or the Java code to extract the primitive attributes such as $action$ and $category$. Then, to extract extra parameters, StoryDistiller identifies the methods related to the activity life cycle and analyzes these methods successively based on the relation between the additional parameters in these methods and activity rendering.

Finally, StoryDistiller launches activities dynamically using the extracted ICC data with the help of the Android Debug Bridge (ADB) and the Android Emulator. Then, StoryDistiller explores all interactive components of each activity to identify the dynamic activity transitions. The dynamic activity transitions are merged with the static activity transitions to generate the final ATG. During this process, StoryDistiller also takes screenshots of each activity in the app. As a result, StoryDistiller can generate the ATG of the app, together with the screenshots of each activity in the app.

Note that the execution of StoryDistiller is time-consuming. To help users save time, we cache the ATG of each app that has been uploaded to our server. More specifically, if there is a new app to analyze, we first check whether the ATG of this app has been cached. If so, we directly return the ATG to the user. Otherwise, we analyze the app using StoryDistiller and cache the ATG of this app for future use.

C. Debloating Android Application

After collecting the activities that the user does not want to keep, we need to remove these activities from the app. To do so, we first identify the methods in the classes of the activities that the user wants to remove (i.e., to-remove classes). Then we identify other related methods in the CG that are only affected by these methods using forward slicing. More specifically, we first annotate the methods that are in the to-remove classes as the initial set of to-remove methods. We copy the to-remove methods to a worklist. We first pop a method from the worklist and identify the successors of this method in the CG. Then, for each successor, if it is only called by the methods in the to-remove methods, we add the successor to the to-remove methods and the worklist. The algorithm terminates when the worklist becomes empty, indicating that all relevant nodes have been processed.

We employ the Soot framework [10] to eliminate the to-remove methods from the Android application. Each to-remove method undergoes a process where its body is cleared, and the return value is modified to either `null` (for reference types in Java) or 0 (for numeric types in Java). By doing so, these to-remove methods are effectively excluded from being called, while still maintaining their presence to ensure program compilation.

We have developed a Java library that incorporates the method-removal module. This library takes as input the original application, its corresponding call graph (CG), and the activities to be removed. It then generates a debloated version of the application as its output.

III. Evaluation

To demonstrate the usefulness of AutoDebloater, we conduct a user study. Our goal is to check whether AutoDebloater can help explore the functionalities of apps effectively, and can help users remove the activities that they are unlikely to use.

We randomly select five apps (i.e., Bitcoin Wallet[5], Amaze File Manager[6], Gas Prices[7], Vespucci[8], and A2DP Volume[9]) with different numbers of activities (3-19 activities) from three categories (i.e., finance, tool and navigation), which are hosted on Google Play Store and F-Droid.

We invite seven users to assess the performance of AutoDebloater. All of the recruited participants have used Android devices for more than one year and are not Android-related developers or researchers. We first explain the concept of debloating to them, to ensure that they understand the goal of their task. Then, we ask them to explore the functionalities of the selected apps and visit our website to browse the ATG of these selected apps. Note that the selected apps are uploaded to the website in advance. Therefore, according to Section II-B, our website has the cache of the ATG and the users can directly obtain the ATG of the apps without waiting for the analysis of the apps. If they find that there are some activities that they will never use, we ask them to remove these activities using AutoDebloater. Finally, we ask each user to assign a score from 1 to 5 to each debloated app in terms of *stability* (i.e., the ability of AutoDebloater to create a debloated app that runs without crashing) and *overall satisfaction* (i.e., the ability of AutoDebloater to help them identify activities that they are unlikely to use and to remove them). 1 indicates that the user is not satisfied at all, and 5 indicates that the user is very satisfied.

Our user study result uncovers that, the average score of *stability* is 4.8, and the average score of *overall satisfaction* is 3.97. This indicates that users are satisfied with the AutoDebloater as well as the debloated apps. Some users complain

[4]https://developer.android.com/guide/fragments

[5]https://play.google.com/store/apps/details?id=de.schildbach.wallet

[6]https://play.google.com/store/apps/details?id=com.amaze.filemanager

[7]https://f-droid.org/en/packages/org.woheller69.spritpreise/

[8]https://f-droid.org/en/packages/de.blau.android/

[9]https://play.google.com/store/apps/details?id=a2dp.Vol

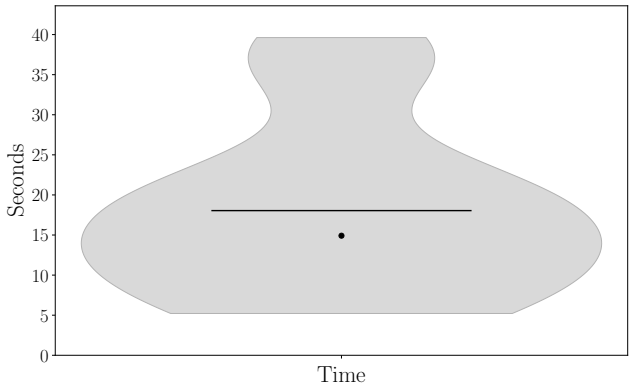

Fig. 3. The distribution of the time to debloating. This figure shows that the time to debloating in AutoDebloater is acceptable.

that activities can still not be fully rendered in ATG, which is already reported in Chen et al.'s work [9]. Our website also records the time to debloat. Figure 3 shows the plot of the time to debloat. The dot in the plot indicates the median time to debloat the app, and the horizontal line indicates the average time to debloat. On average, the time to debloat is less than 20 seconds, which we believe to be acceptable.

IV. Related Work

Google has provided a series of off-the-shelf tools to debloat Android apps. However, the goal of these tools is to reduce the size of Android apps rather than remove the activities that users will never use. For example, R8 is used to statically detect and remove dead code (e.g., unused classes, fields, methods) and unused resources from an app.[10,11] Google Play also provides the App Bundle mechanism that only the code and resources of a specific device could be downloaded [11].

Researchers also proposed approaches to reduce the size of Android apps. For example, Jiang et al. identified and removed the redundant bytecode from the app statically [12]. Following that, Jiang et al. also employed static analysis to detect and remove dead code [5]. Pilgun et al. considered the code that is not executed during tests as bloated code and removed the bloated code from Smali code [6]. Xie et al. debloated apps to minimize the bandwidth of mobile networks [13].

The most related work to our paper is the work of Tang et al. [7]. Different from these works, Tang et al. considered the features (i.e., activity, permission, and modularity) in the apps that can be bloated and asked developers to annotate the bloated features [7]. They also removed unused resources, images, and ABI from the app. However, when we submit our paper, the code of their work is not available.[12] Besides, different from Tang et al.'s work, our paper implements an off-the-shelf tool to debloat Android apps in the form of a website. Our tool can help users explore the functionalities of apps and remove the activities that they will never use without the help of developers.

[10]https://r8.googlesource.com/r8

[11]https://android-developers.googleblog.com/2018/11/r8-new-code-shrinker-from-google-is.html

[12]https://web.archive.org/web/20230525171119/https://sites.google.com/view/xdebloat We cannot download the code of their work.

V. Conclusion and Future Work

In this paper, we propose a website, AutoDebloater, to debloat Android apps. AutoDebloater can help users explore the functionalities of apps with the help of StoryDistiller and can help users remove the activities that they will never use. Our user study results show that users are satisfied with AutoDebloater in terms of the stability of the debloated apps and the ability of AutoDebloater to identify features that are never noticed before. In the future, we plan to allow users to label the features that they do not want in a finer granularity (e.g., the granularity of a button or a text field).

Acknowledgements

This research / project is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity Research and Development Programme, NCRP25-P03-NCR-TAU. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] "Android," Mar. 2023. [Online]. Available: https://frida.re/docs/android/

[2] "Android Mobile App Developer Tools." [Online]. Available: https://developer.android.com/

[3] A. Aijaz and C. Jang, "The 80% Rule of Software Development," Feb. 2020. [Online]. Available: https://www.split.io/blog/the-80-rule-of-software-development/

[4] S. Bhattacharya, K. Gopinath, and M. G. Nanda, "Combining concern input with program analysis for bloat detection," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 745–764, Nov. 2013.

[5] Y. Jiang, Q. Bao, S. Wang, X. Liu, and D. Wu, "RedDroid: Android Application Redundancy Customization Based on Static Analysis," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2018, pp. 189–199.

[6] A. Pilgun, "Don't Trust Me, Test Me: 100% Code Coverage for a 3rd-party Android App," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. Singapore, Singapore: IEEE, Dec. 2020, pp. 375–384.

[7] Y. Tang, H. Zhou, X. Luo, T. Chen, H. Wang, Z. Xu, and Y. Cai, "XDebloat: Towards Automated Feature-Oriented App Debloating," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4501–4520, Nov. 2022.

[8] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "StoryDroid: Automated Generation of Storyboard for Android Apps," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 596–607.

[9] S. Chen, L. Fan, C. Chen, and Y. Liu, "Automatically Distilling Storyboard With Rich Features for Android Apps," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 667–683, Feb. 2023.

[10] "BodyTransformer (Soot API)." [Online]. Available: https://www.sable.mcgill.ca/soot/doc/soot/BodyTransformer.html

[11] "About Android App Bundles." [Online]. Available: https://developer.android.com/guide/app-bundle

[12] Y. Jiang, D. Wu, and P. Liu, "JRed: Program Customization and Bloatware Mitigation Based on Static Analysis," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Atlanta, GA, USA: IEEE, Jun. 2016, pp. 12–21.

[13] Q. Xie, Q. Gong, X. He, Y. Chen, X. Wang, H. Zheng, and B. Y. Zhao, "Trimming Mobile Applications for Bandwidth-Challenged Networks in Developing Regions," *IEEE Transactions on Mobile Computing*, vol. 22, no. 1, pp. 556–573, Jan. 2023.