Towards Speedy Permission-Based Debloating for Android Apps

Ferdian Thung[1], Jiakun Liu[1], Pattarakrit Rattanukul[1], Shahar Maoz[2], Eran Toch[3], Debin Gao[1], and David Lo[1]
{ferdianthung,jkliu,pattarakritr,dbgao,davidlo}@smu.edu.sg,maoz@cs.tau.ac.il,erant@tauex.tau.ac.il
[1]School of Computing and Information Systems, Singapore Management University, Singapore
[2]School of Computer Science, Tel Aviv University, Israel
[3]Department of Industrial Engineering, Tel Aviv University, Israel

ABSTRACT

Android apps typically include many functionalities that not all users require. These result in software bloat that increases possible attack surface and app size. Common functionalities that users may not require are related to permissions that they intend to disallow in the first place. As these permissions are disallowed, their related code would never be executed and therefore can be safely removed. Existing work has proposed a solution to debloat Android apps according to the disallowed permissions. However, for large and complex applications, the debloating process could take hours, typically due the long time that may be needed to construct call graph for analysis. In this work, we propose MiniAppPerm, that speeds up the permission-based debloating by constructing a partial call graph instead of a complete call graph. Our preliminary experiments on a set of apps in Google Play show that MiniAppPerm can reduce the call graph construction time by up to 85.3%. We also checked that the debloated apps can run without crashes.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

ACM Reference Format:
Ferdian Thung[1], Jiakun Liu[1], Pattarakrit Rattanukul[1], Shahar Maoz[2], Eran Toch[3], Debin Gao[1], and David Lo[1]. 2024. Towards Speedy Permission-Based Debloating for Android Apps. In *IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MOBILESoft '24), April 14–15, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3647632.3651390

1 INTRODUCTION

Modern Android apps contain a lot of features to support different needs of their target users. These include but not limited to different use case scenarios, different versions of libraries, and various application binary interface (ABI) for various device architectures [7]. Due to different user needs, some features can be safely removed as users never need them. Features that users commonly do not need are the ones related to permissions that they disallow. Since the permissions are disallowed, the features would not work. Thus, we can safely remove all features related to the disallowed permissions.

Several works have explored Android apps debloating/pruning [7, 8, 14], with XDebloat [14] as the most recent work that can perform permission-based debloating. These work mainly focused on realizing the capability of debloating Android apps. While they did measure the execution time, it is not their main goal. They typically rely on common tools such as FlowDroid to construct call graph and perform static program analysis [1]. While this saves development time, they would also inherit FlowDroid shortcomings. It has been reported that FlowDroid call graph generation time could take many hours [3, 15] for modern apps, which would lengthen the overall execution time of permission-based debloating solutions. This is not desirable as users would need to wait longer to receive the debloated apps. Thus, there is a need for a faster permission-based debloating, which can reduce users' waiting time.

In this work, we propose MiniAppPerm to speed up permission-based debloating by constructing a partial call graph instead of a complete one. When we debloat features related to permissions, we need to remove the permission-related methods (i.e., Android APIs that require the specific permission) that are called by the app. We then need to traverse the call graph to find other methods that may be affected by the removal of these permission-related methods. To do so, we do not really need to explore the whole call graph. We only need to explore the methods that are reachable from the permission-related methods. Thus, we do not actually need to construct the portion of call graph that are not reachable from permission-related methods, as they would not be used and waste computation time. Constructing a partial call graph with methods reachable from permission-related methods should suffice.

Since MiniAppPerm constructs a partial call graph, the call graph construction time should be faster. Intuitively, the amount of time saved is commensurable to the proportion of unreachable methods in the complete call graph. To measure how much time MiniAppPerm can actually save when constructing call graph, as a preliminary experiment, we collected a dataset of modern Android apps. We first obtain the list of popular apps from Android Rank[1], which is a website that collects historical Android data and Google Play[2] since 2011. We then downloaded the apps in the list from Google Play. We sample some apps for preliminary experiments. We ran FlowDroid on these apps and collected their call graph construction time. Our experiments on these apps highlight that MiniAppPerm can successfully reduce the call graph construction time by up to

[1]https://www.androidrank.org/
[2]https://play.google.com/

ACM ISBN 979-8-4007-0594-6/24/04
https://doi.org/10.1145/3647632.3651390

85.3%. We have also manually checked that the resultant debloated apps can be successfully installed and ran without crashes.

2 MINIAPPPERM

The overview of MiniAppPerm is presented in Figure 1. It takes an Input App and the list of Disallowed Permissions and outputs Debloated App. It has four components: (1) Permission Identifier; (2) Call Graph Constructor; (3) Call Graph Slicer; and (4) Debloater. Permission Identifier reads the list of Disallowed Permissions and identifies the Permission-Related Methods from the Input App. Call Graph Constructor constructs a Partial Call Graph containing only methods reachable from the Permission-Related Methods. Call Graph Slicer traverses the Partial Call Graph and identifies the methods that can be removed due to removal of Permission-Related Methods. Last but not least, Debloater takes the list of Methods to Remove and returns the Debloated App, which is a repackaged Input App after discarding the Methods to Remove.

Permission Identifier. To identify whether permission-related methods are called in the app, it makes use of a known mapping between a permission and permission-related methods (i.e., Android APIs that require the specified permission). We take the mapping from PScout [2], which produces the mapping by performing a reachability analysis between permission checks and API calls in Android framework. Given the list of Disallowed Permissions, it obtains the corresponding permission-related methods. For example, `setAudioSource(int)` method requires `RECORD_AUDIO` permission. It collects such methods for all Disallowed Permissions. Given Input App, it dumps the bytecode of Input App into a plaintext using `dexdump`[3]. It iterates the collected permission-related methods and check whether each of them is invoked in the bytecode plaintext. It then returns the Invoked Permission-Related Methods.

Call Graph Constructor. It is built on top of BackDroid, which is proposed by Wu et al. [15]. BackDroid performs a targeted inter-procedural analysis that can skip unreachable code and follow only the paths that lead to security-sensitive sink APIs. Similarly, to build the Partial Call Graph, it adopts the targeted inter-procedural analysis and treats the Invoked Permission-Related Methods as the sink APIs. It performs a targeted backtracking from the Invoked Permission-Related Methods. For each invoked permission-related method, it searches the bytecode plaintext of the Input App for the call sites of the corresponding permission-related method. For each call site, it identifies the caller of permission-related method. It repeats the search until it reaches callers that are Android entry points (e.g., Android lifecycle methods).

Call Graph Slicer. Given the Partial Call Graph, it first puts the Invoked Permission-Related Methods to the list of Methods to Remove. It then traverses the call graph to find methods that would be affected by the removal of Invoked Permission-Related Methods. Specifically, it performs two kinds of slicing: (1) *Forward slicing*. It recursively put methods that are only called by methods in the list of Methods to Remove into the list of Methods to Remove. The intuition is that such methods would never be called since all their callers would be removed; (2) *Backward slicing*. It recursively put methods that only call methods in the list of Methods to Remove into the list of Methods to Remove. The intuition is that such methods do not have much functionality as all the methods they called are removed. After it finds all methods that satisfy the above conditions, it returns the final Methods to Remove to the next component for debloating.

[3]https://manpages.ubuntu.com/manpages/bionic/man1/dexdump.1.html

Table 1: Statistics of the Studied Android Apps

ID	App	Category	#Perm	#Install
1	org.wikipedia	Books & Reference	13	50M+
2	com.apusapps.browser	Communication	22	10M+
3	com.halamate.app	Lifestyle	28	5M+
4	com.sourceapp.ebb	Food & Drink	10	500K+
5	com.canva.editor	Art & Design	11	100M+
6	com.lovense.wear	Health & Fitness	37	1M+
7	com.nytimes.cooking	Food & Drink	7	100K+
8	com.amomedia.madmuscles	Health & Fitness	9	500K+
9	com.ecw.healow	Health & Fitness	22	5M+
10	com.tacobell.ordering	Food & Drink	20	10M+

Debloater. Debloater performs a method-level debloating. For each Method to Remove, Debloater discards the content of the method body. If the method has a return value, Debloater changes the return value according to its type. If the return type is a numeric class, Debloater changes the return value to 0. If the return type is an instance of other classes, Debloater changes the return value to null. Debloater then repackaged the modified Input App and outputs it as the Debloated App.

3 PRELIMINARY EXPERIMENT

Dataset. We collected modern Android apps from the list of popular apps taken from Android Rank[4] on April 2023. This website provides a list of top Android apps. The apps are divided into 33 categories such as Art and Design, Business, Productivity, Sport, etc. We downloaded the list of top apps from each category and obtained the latest version of the apps from APKCombo[5], which is a third party website that pulls the apps directly from Google Play and make them downloadable from their website. We then sample 10 apps for preliminary experiments. Note that we exclude apps that crash when we run it through FlowDroid, BackDroid, or Soot. Statistics of the studied apps are shown in Table 1.

Experimental Settings. We used the latest numbered version of FlowDroid at the time (v2.111.1) and construct call graph for the studied apps. We found that different versions of FlowDroid could have a significant impact on the call graph construction time. When constructing the call graph, we run FlowDroid for up to 5 hours following Wu et al. [15]. Similarly, we also run MiniAppPerm for the same amount of time and measure how long it takes to construct the call graph. We consider the worst case scenario where we disallow all permissions. We then compare the call graph construction time between FlowDroid and MiniAppPerm. We measure how much call graph construction time MiniAppPerm can reduce. We also measure the overall debloating time of MiniAppPerm.

It is worth noting that we cannot compare MiniAppPerm with existing debloating approaches (e.g., XDebloat [14]) since they do not release their tools. Nevertheless, XDebloat is built on top of

[4]https://www.androidrank.org/
[5]https://apkcombo.com/

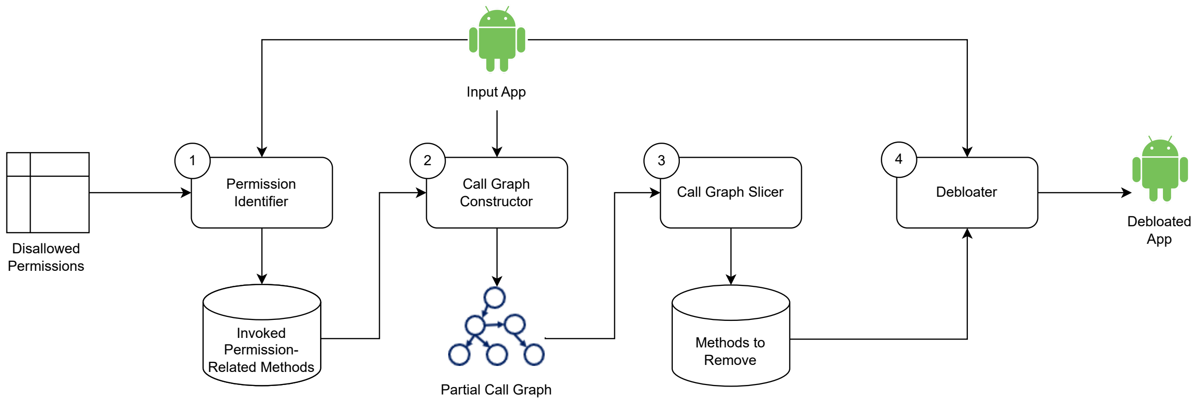

Figure 1: Overview of MiniAppPerm

Table 2: Debloating and call graph construction time

ID	Debloating Time	Call Graph Construction Time		
		FlowDroid	MiniAppPerm	Impr.
1	1m 43s	2m 23s	1m 30s	37.1%
2	8m 31s	10m 6s	8m 10s	19.1%
3	2m 34s	5m 13s	46s	85.3%
4	7m 24s	8m 9s	6m 22s	21.9%
5	7m 29s	7m 20s	7m 5s	3.4%
6	3m 5s	8m 10s	1m 25s	82.7%
7	6m 2s	7m 21s	5m 41s	22.7%
8	6m 33s	10m 21s	6m 13s	39.9%
9	13m 15s	12m 41s	12m 15s	3.4%
10	14m 38s	15m 1s	14m 14s	5.2%

FlowDroid and thus its call graph construction time should be similar with running FlowDroid directly. Additionally, MiniAppPerm also does not compete with existing permission-based debloating approaches. Rather, it complements them.

Results. Table 2 shows the total debloating and call graph construction time when running FlowDroid and MiniAppPerm on the studied apps. The total debloating time measures the total time required for debloating the app starting from the Input App and ending with the Debloated app. We observe that, for most apps, the total debloating time is dominated by the call graph construction time, even after the speed up achieved by MiniAppPerm. This shows that, for most apps, the call graph construction time takes up the majority of execution time and it is indeed the right target for optimizing total debloating time.

We can see from the table that MiniAppPerm can reduce the the call graph construction time from 3.4% to 85.3%. The highest reductions are achieved when debloating `com.halamate.app` and `com.lovense.wear`. Interestingly, these two apps have the highest number of permissions among the studied apps. On the other hand, the lowest reductions are achieved when debloating `com.canva.editor` and `com.ecw.healow`. These two apps have a very different number of permissions. Therefore, it appears that the call graph construction time does not depend on the number of permissions. Indeed, the time likely depends more on how widespread the permission-related methods are being called within the app.

4 RELATED WORK

Many works have studied program debloating [6, 8, 9, 11–14]. Heo et al. [6] proposed Chisel to debloat C programs. Chisel takes a program to debloat and high-level specification of its desired functionality, and outputs the debloated program that follows the specification. Quach et al. [11] proposed an approach that can perform piece-wise compilation and loading, which can systematically detect and eliminate unused code from memory. Rastogi et al. [12] developed Cimplifier, which debloats containers (i.e. Docker) based on user-defined constraints. Sharif et al. [13] proposed TRIMMER, which debloats unused functionality from C programs by specializing the programs according to the deployment context. Jiang et al. [9] proposed JRed to trim unused code from Java applications and Java Runtime Environment (JRE) via static analysis. In a subsequent work, Jiang et al. [8] proposed RedDroid, which debloats Android applications by removing compile-time and install-time redundancies. More recently, Tang et al. [14] proposed XDebloat to perform feature-oriented debloating for Android apps. Our work complements the above work by speeding up the debloating process through optimizing the call graph construction.

Some works aim to detect and analyse software bloat [4, 5, 17]. Bhattacharya et al. [4] proposed an approach to detect an execution bloat in Java applications by utilizing concern information. Xu et al. [17] analysed how one can find, remove, and prevent performance problems due to software bloat. Bu et al. [5] highlighted the importance of bloat-aware design in developing Big Data applications. Nguyen et al. [10] analyses the impact of software bloat in data-intensive systems. Xu et al. [16] introduced an abstraction called copy graph that can be used to expose common patterns of bloat in Java. The above studies provides foundations for work on debloating approaches. One should detect a bloat first before removing it. Better understanding on software bloat is also beneficial in the development of future debloating approaches.

5 CONCLUSION AND FUTURE WORK

We propose MiniAppPerm to speed up permission-based debloating for Android apps. It does so by constructing a partial call graph. Our preliminary evaluation indicates that MiniAppPerm can reduce the call graph construction time by up to 85.3%. In the future, we plan to ascertain MiniAppPerm effectiveness more extensively on more apps. We also plan to improve MiniAppPerm speed by optimizing the partial call graph construction, e.g., by introducing parallelization. Another possibility is to perform on-the-fly slicing without fully constructing the call graph. Yet another direction is to employ neurosymbolic program analysis to construct the partial call graph, e.g., by predicting the edges of the partial call graph using deep learning techniques.

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. This work is also partly supported by the Blavatnik Interdisciplinary Cyber Research Center at Tel Aviv University as part of a collaboration with CSA Singapore, and Lee Kong Chian Fellowship awarded to Debin Gao by Singapore Management University.

REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[2] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 217–228.

[3] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining apps for abnormal usage of sensitive data. In *2015 IEEE/ACM 37th IEEE international conference on software engineering*, Vol. 1. IEEE, 426–436.

[4] Suparna Bhattacharya, Kanchi Gopinath, and Mangala Gowri Nanda. 2013. Combining concern input with program analysis for bloat detection. *ACM SIGPLAN Notices* 48, 10 (2013), 745–764.

[5] Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J Carey. 2013. A bloat-aware design for big data applications. In *Proceedings of the 2013 international symposium on memory management*. 119–130.

[6] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 380–394.

[7] Jianjun Huang, Yousra Aafer, David Perry, Xiangyu Zhang, and Chen Tian. 2017. UI driven Android application reduction. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 286–296.

[8] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. 2018. RedDroid: Android application redundancy customization based on static analysis. In *2018 IEEE 29th international symposium on software reliability engineering (ISSRE)*. IEEE, 189–199.

[9] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. Jred: Program customization and bloatware mitigation based on static analysis. In *2016 IEEE 40th annual computer software and applications conference (COMPSAC)*, Vol. 1. IEEE, 12–21.

[10] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, and Guoqing Xu. 2018. Understanding and combating memory bloat in managed data-intensive systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 26, 4 (2018), 1–41.

[11] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piece-wise compilation and loading. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 869–886.

[12] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 476–486.

[13] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 329–339.

[14] Yutian Tang, Hao Zhou, Xiapu Luo, Ting Chen, Haoyu Wang, Zhou Xu, and Yan Cai. 2021. Xdebloat: Towards automated feature-oriented app debloating. *IEEE Transactions on Software Engineering* 48, 11 (2021), 4501–4520.

[15] Daoyuan Wu, Debin Gao, Robert H Deng, and Chang Rocky KC. 2021. When program analysis meets bytecode search: Targeted and efficient inter-procedural analysis of modern Android apps in BackDroid. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 543–554.

[16] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the flow: Profiling copies to find runtime bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 419–430.

[17] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 2010. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. 421–426.