Poster: Kerberoid: A Practical Android App Decompilation System with Multiple Decompilers

Heejun Jang
Sungkyunkwan University
hj.jang@skku.edu

Beomjin Jin
Sungkyunkwan University
jinbumjin@skku.edu

Sangwon Hyun
Myongji University
shyun@mju.ac.kr

Hyoungshick Kim
Sungkyunkwan University and CSIRO Data61
hyoung@skku.edu

ABSTRACT

Decompilation is frequently used to analyze binary programs. In Android, however, decompilers all perform differently with varying apps due to their own characteristics. Obviously, there is no universal solution in all conditions. Based on this observation, we present a practical Android app decompilation system (called Kerberoid) that automatically stitches the results from multiple decompilers together to maximize the coverage and the accuracy of decompiled codes. We evaluate the performance of Kerberoid with 151 Android apps in which their corresponding source codes are publicly available. Kerberoid fully recovered all functions for 17% of the apps tested and gained a similarity score over 50% for 40% of the apps tested, increased by 7% and 9%, respectively, compared with the best existing decompiler.

CCS CONCEPTS

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

KEYWORDS

Decompilation; Android apps; Mobile security; Reverse engineering

ACM Reference Format:
Heejun Jang, Beomjin Jin, Sangwon Hyun, and Hyoungshick Kim. 2019. Poster: Kerberoid: A Practical Android App Decompilation System with Multiple Decompilers. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3319535.3363255

1 INTRODUCTION

Decompilation is a common technique to analyze a binary program by transforming it into high-level source codes that can be easily understandable. In Android, many decompilation methods have also been introduced to analyze the behaviors of apps (e.g., detecting third party libraries [1] in apps). However, the performance of existing decompilers is still limited in real world environments [2] because code optimization and obfuscation techniques may cause a loss of data which is necessary for recovering original source codes in the decompilation process. Interestingly, existing decompilers all perform differently with varying apps. There is no single winner that always produces the best decompilation results in all conditions. Each decompiler has its own advantages and disadvantages. For example, when the Android app, Aegis (https://github.com/beemdevelopment/Aegis), is given for decompilation, Jadx (https://github.com/skylot/jadx) and JD Project (http://java-decompiler.github.io/) decompilers successfully recovered the `convertEntry` function while CFR (http://www.benf.org/other/cfr/) decompiler failed. On the other hand, CFR recovered the `read` function while the other decompilers failed.

Based on this observation, we propose the idea of Kerberoid[1] that properly merges partial results obtained from multiple decompilers into a final decompilation result to improve the coverage and the accuracy of decompiled codes. To show the feasibility of Kerberoid, we specifically implement Kerberoid with three representative Android decompilers (CFR, Jadx and JD Project).

To evaluate the performance of Kerberoid, we collect 151 Android apps whose corresponding source codes are publicly available and run Kerberoid with those apps. The evaluation results show that Kerberoid outperformed Jadx, CFR and JD Project with respect to the number of recovered functions and the similarity score between the original source codes and the decompiled codes.

2 OVERVIEW OF KERBEROID

Kerberoid is comprised of four components: (1) *Collector*, (2) *Parser*, (3) *Integrator*, and (4) *Selector*. Figure 1 shows the overview of Kerberoid. Kerberoid takes an Android Application Package (APK) file as input and performs the following steps to generate its decompiled Java codes with *external* decompilers.

- ***Collector*** initially runs several external Java decompilers with an input APK file and then collects their outputs (i.e., decompiled Java codes from those decompilers)[2]. The decompiled codes from each decompiler are organized according to a predefined structure and then passed to *Parser*. In our prototype implementation,

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11.
https://doi.org/10.1145/3319535.3363255

[1] Based on the name (*Kerberos*) of a three-headed dog in Greek mythology.
[2] Before running decompilers, dex2jar tool (https://github.com/pxb1988/dex2jar) is used to convert the input APK file to the corresponding Java archive (JAR) file because each Java decompiler typically takes a JAR file as input.

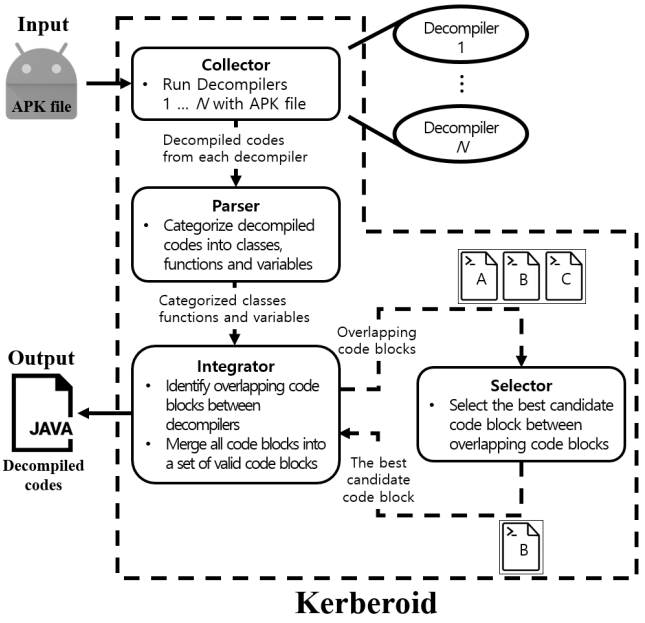


Figure 1: Overview of Kerberoid.

we specifically use CFR, Jadx and JD Project decompilers because those are popularly used ones.

- ***Parser*** categorizes decompiled codes into types of classes, functions and variables. This categorization process can be implemented by observing *unique* start and end tags of each code type. For example, the source code of a class starts with the keyword "class" and ends with the closing brace "}." The categorized code blocks (i.e., classes, functions and variables) are passed to *Intergrator*. The code blocks such as classes and functions are sorted by name to speed up the process in *Intergrator* for comparing classes, functions and variables with their names between Java source codes produced by different decompilers.
- ***Intergrator*** assembles the code blocks obtained from *Parser* to form into the final decompiled codes for the input APK file by removing overlapping code blocks generated by two or more decompilers. For this task, *Intergrator* compares the code blocks produced by a decompiler with the code blocks produced by the other decompilers. In our prototype implementation, such overlapping code blocks can be identified with name, parameter/return type, and number of parameters. When overlapping code blocks are identified, those code blocks are passed to *Selector*, which in then selects and recommends the most suitable one of the overlapping code blocks to *Integrator*. We found that for over 88.2% of all the decompiled functions in our experiments overlapping code blocks were generated from multiple decompilers thus *Selector* was triggered quite frequently.
- ***Selector*** compares overlapping code blocks and selects the best candidate code block among them using a multi-class classification model. The selected best candidate code block is finally recommended to *Intergrator*. In our prototype implementation, we built a random forest classifier with the following three features. The first feature is the number of code lines making up each decompiled code block (e.g., function or class). We found that the shorter the length of each decompiled code block, the higher the similarity between the decompiled code block and the corresponding original code block. The second feature is the number of parameters used for a function. Interestingly, the best decompilation results for a function were somewhat inconsistent with the number of parameters. When the number of parameters is less than 4, Jadx typically produced the best results in terms of similarity with the original code blocks. However, in the other cases, JD project outperformed the other decompilers. The third feature is the number of loop (e.g., `for` and `while`) or conditional instructions (e.g., `if` and `switch`). The performance of decompilers can be highly varied depending on the inclusion of those instructions. Based on those three features, we construct a classifier to recommend the best candidate block.

Our Kerberoid prototype is implemented in Python 2.7 and tested on Linux Ubuntu 16.04 system. The source code is available on GitHub (https://github.com/ljyjhj/Kerberoid).

3 EVALUATION

This section presents our evaluation of Kerberoid against existing decompilers with respect to the coverage and the accuracy of decompiled codes.

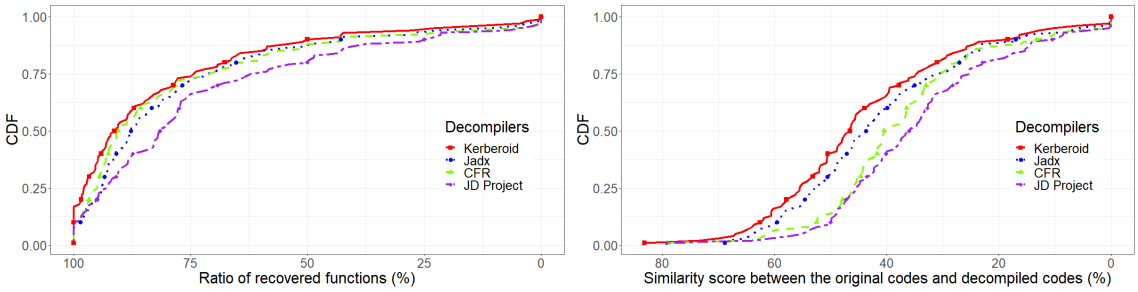
3.1 Experiment setup

We collected the APK files and the corresponding source code files of 151 apps from Fossdroid website (https://fossdroid.com/), a public repository of open source Android apps. We divided 151 apps into a training set (one-third of the apps) and a testing set (the remaining apps) in which the training set is used to build *Selector* for Kerberoid and the testing set is used to evaluate its performance. For performance comparison, we also tested Jadx, CFR and JD Project on the same set of APK files and obtained the decompiled codes from them.

To evaluate the effectiveness of Kerberoid, we need to observe how much portion of codes it can successfully recover and also how similar the recovered codes are compared to the original source codes. As a means to see the code coverage of Kerberoid, we just counted the number of functions that have been successfully recovered by it and compared the results from other decompilers.

In addition, we calculated the similarity score (ranging from 0% to 100%) between the original source codes and the decompiled codes from Kerberoid. To calculate the similarity score between two source codes, ssdeep (https://ssdeep-project.github.io/ssdeep/index.html) was used because ssdeep is known as a reliable fuzzy hash algorithm for checking the similarity between codes [3].

Because an Android app typically consists of several Java source files, we need to extend the similarity score for a single source file to the similarity score for the entire app consisting of multiple source files. To calculate the similarity score for the entire app, we can simply use a weighted sum of similarity scores for all files in the app. That is, for each Java source file, we first calculate the similarity score, weighted by the ratio of the file size to the total file size of the app, and then summed them all to calculate the final similarity score for the app. We also compared the similarity score of Kerberoid with those from other decompilers.



(a) Ratio of successfully recovered functions. **(b) Similarity score to the original codes.**

Figure 2: Performance of decompilers.

3.2 Evaluation results

We evaluate the performance of Kerberoid against Jadx, CFR and JD Project with respect to two metrics: (1) the number of successfully recovered functions and (2) the similarity score between the original source codes and the decompiled codes (see Figure 2).

Figure 2a shows the cumulative distribution functions (CDFs) of the ratio of successfully recovered functions by each decompiler in the tested apps. Overall, the results produced by Kerberoid are better than those produced by other decompilers – Kerberoid showed over 75% recovery ratio of functions for about 75% of all the tested apps.

We also compared the ratio of apps where all their functions were fully recovered and the number of apps where no function was recovered at all. As shown in Table 1, Kerberoid fully recovered all functions in 17% of the apps tested, showing significant superiority against other decompilers (8–10%).

Table 1: Ratio of apps where all the functions were fully recovered (Full) and ratio of apps where no function was recovered at all (Failed) by each decompiler.

Result	Kerberoid	Jadx	CFR	JD Project
# Full	**17%**	8%	10%	10%
# Failed	**2%**	3%	4%	4%

Figure 2b shows the CDFs of the similarity scores between the original codes and decompiled codes by each decompiler with the tested apps. Again, these results showed superiority of Kerberoid over other decompilers with respect to the similarity score. Table 2 shows the detailed numbers of the apps decompiled by each decompiler, respectively, with over 40%, 50% and 60% similarity score. Kerberoid decompiled more than 40% of the apps tested with a similarity score higher than 50% while other decompilers recovered 9–31% apps in the same condition.

Table 2: Ratio of apps with over 60%, 50%, and 40% similarity score by each decompiler.

Result	Kerberoid	Jadx	CFR	JD Project
# Over 60%	**16%**	9%	6%	3%
# Over 50%	**40%**	31%	13%	9%
# Over 40%	**65%**	59%	51%	40%

Interestingly, Table 1 and 2 show that each decompiler has its own advantages and disadvantages – CFR and JD Project achieve higher code coverage (10% for '# Full') while Jadx achieves higher similarity to the original source codes (31% for '# Over 50%'). We note that such characteristics of decompilers may explain the reason why Kerberoid can produce the best decompilation results.

4 CONCLUSION

We propose a novel Android app decompilation system named Kerberoid that achieves synergy between multiple decompilers by complementing the results from those decompilers each other to maximize the coverage and the accuracy of decompiled codes. Specifically, we developed a machine learning-based classifier to find the best code block when there are multiple candidate code blocks generated from multiple decompilers for the same function.

To validate the feasibility of Kerberoid, we implemented a prototype of Kerberoid and performed experiments with 151 Android apps. In the experiments, Kerberoid outperformed the existing decompilers (Jadx, CFR and JD Project) with respect to the number of recovered functions and the similarity score between the original source codes and the decompiled codes. Kerberoid fully recovered all functions for 17% of the apps tested while the second best solutions (CFR and JD Project) recovered all functions for only 10% of the apps. Also, Kerberoid gained similarity scores over 50% for 9% more apps than the second best solution (Jadx).

As part of future work, we plan to extend Kerberoid to effectively decompile even obfuscated codes with deobfuscators.

ACKNOWLEDGMENTS

This research was supported by the ITRC program (IITP-2019-2015-0-00403) and IITP grant funded by the Korea government (MSIT) (No.2018-0-00532, Development of High-Assurance (≥EAL6) Secure Microkernel).

REFERENCES

[1] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016.

[2] Vitor Monte Afonso, Paulo L. de Geus, Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, Giovanni Vigna, Adam Doupé, and Mario Polino. Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, 2016.

[3] Yan Wang, Haowi Wu, Hailong Zhang, and Atanas Rountev. Orlis: Obfuscation-Resilient Library Detection for Android. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, 2018.