Poster: ART-assisted Android App Diffing

Jakob Bleier
TU Wien, Vienna, Austria
jakob.bleier@tuwien.ac.at

Martina Lindorfer
TU Wien, Vienna, Austria
martina@iseclab.org

***Abstract*—Android aims to provide a secure and feature-rich, yet resource-saving platform for its applications (apps). To achieve these goals, the compilation to distributable packages shrinks, obfuscates, and optimizes the code by default. As an additional optimization, the Android Runtime (ART) nowadays compiles the app's bytecode to native code on the device instead of executing it in the Dalvik VM. We study the effects of these changes in the Android build and runtime environment on the problem of calculating app similarity. We compare existing bytecode-based tools to our novel approach of using the re-compiled (and optimized) binary form.**

We propose OATMEAL, an extensible framework to generate reliable ground truth for evaluating app similarity approaches and provide a benchmark dataset to the community. We built this dataset from open-source apps available on F-Droid in various configurations that optimize and obfuscate the bytecode.

Using this dataset, we show the limitations of existing Android-specific bytecode analysis approaches when faced with the new optimizing R8 bytecode compiler. We further demonstrate how well BinDiff, a state-of-the-art binary-based alternative, works in scoring the similarity of apps. With OATMEAL, we provide the foundation for integrating and benchmarking further approaches, both for calculating the similarity between apps (based on bytecode or binary code), and for evaluating their robustness to evolving optimization and obfuscation techniques.

I. Evolution of the Android Build Toolchain

The Android operating system has seen many changes with its yearly release cycle. One of the biggest is the replacement of dx with the R8 [6] compiler, which creates Dalvik bytecode from Java or Kotlin sources. While ProGuard [7] has been supported as obfuscation and optimization pass for a long time, the change has made it obsolete, incorporating its mechanisms in the compiler and enabling them by default for new projects. R8 also adds more optimizations and code shrinking to the build process. The latter removes unused code from not just the app, but also its included libraries.

Furthermore, with the Android Runtime (ART) [5], there is an additional compilation step before apps are executed: It compiles and optimized the Dalvik bytecode to native binary code to speed up execution. Figure 1 shows an overview of the lifetime of an Android app through these various build stages.

These changes in the app build process have repercussions on existing Android code analysis tooling: in our research, we focus on approaches for calculating the similarity between apps. These approaches enable the detection of app clones, malicious impersonification, and changes between versions of an app, such as the application of security patches.

However, besides many approaches no longer being available, or failing to process modern APKs, the changes in the build toolchain break the assumptions they are built on: The assumption that the same method compiles in the same way with the same signature, including its name, in two versions of an app no longer holds for apps compiled with R8.

We surveyed app diffing tools that analyze the code of an app statically, e.g., SimiDroid [11] and Elsim [9] for Dalvik bytecode-based approaches and investigate how they fare on R8-compiled bytecode against binary-based approaches, such as BinDiff [13], applied to ART-compiled apps.

II. Generating a Reliable Dataset

For our study, we needed a dataset of apps with annotated build settings. Ideally, the same apps are compiled with various settings because most apps can be easily unpacked and recompiled. A robust approach to calculating app similarity should be stable regarding various optimization and obfuscation settings, such as working with renamed identifiers.

Due to the lack of existing datasets in this area, we created a benchmark dataset using the F-Droid OSS Market [3]. By extending the build server and hooking into the build process, we created a pipeline to generate reliable build information for real-world apps, modify them, and create ground truth for evaluations. We provide additional processing of APKs with tools such as Redex [4] and ObfuscAPK [1] to study the effects of different optimization and obfuscation settings. Finally, we include tools to automatically compile the apps to binaries, on physical Android phones, emulators, or cross-compilation based on the Android Open Source Project.

We call this pipeline OATMEAL, and it currently produces more than one thousand apps, of which half use R8 features of shrinking, obfuscation, and optimizations.

III. Preliminary Results

In our experiments, we compare the recall and precision of multiple bytecode- and binary-based approaches on sets of apps that have been compiled with the advanced features of R8 en- or dis-abled, as well as further optimizations with Redex. Since these tools typically output a single number between 0 (no similarity) and 1 (complete match), we expect them to be robust against compiler settings and score high for apps that are built from the same source code.

We included SimiDroid [11] and Elsim [2] in our analysis as bytecode-based approaches. For binary-based approaches, we included BinDiff [13] with Ghidra [12] with and IDA Pro [8], as well as Diaphora [10]. Unfortunately, the run time of Elsim

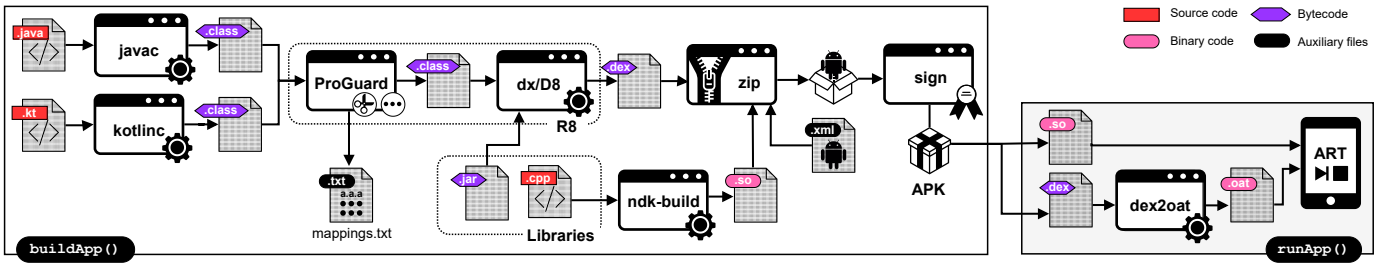

Figure 1: *An Android app from source code to execution:* Java and Kotlin source code and libraries are compiled into optimized and obfuscated Dalvik bytecode. Native libraries written in C/C++ are compiled to binary ELF shared objects. During run time, native libraries are directly loaded by ART, while Dalvik bytecode is compiled into OATs.

(multiple hours per app) and the disk space requirements of Diaphora (multiple gigabytes per app) made them unusable.

Our experiments show, that calculating the binary similarity between apps is comparable to available bytecode-based tools and outperforms them when faced with optimization with Redex. We also find Dalvik-based approaches struggle to process about half of the apps, with or without the new R8 features. In contrast, virtually all apps could be compiled to binaries, and all apps were successfully analyzed.

IV. Future Work

Besides the use case of binary similarity, OATmeal provides all necessary metadata to evaluate library detection on Android apps. Information gathered during the build process can be used to verify which libraries and their versions are used in an app.

To extend the utility of OATmeal, it can be extended to include additional apps from other sources or a curated list of configurations for specific purposes. Because the build server is aware of software repositories and has a powerful configuration notation, any number of additional apps can be integrated.

Last but not least, we only used binary tooling in its default configuration and do not yet analyze the pre-compiled native libraries included in the app. Optimizing settings or including OAT-specific information can be used to improve the efficacy of binary tooling further.

V. Conclusion

We proposed an extensible pipeline to compile apps to OAT binaries, which any available binary tooling can analyze. We selected BinDiff and SimiDroid, two state-of-the-art diffing approaches that operate on binaries and bytecode, respectively, as candidates to demonstrate the utility of our approach.

We required a robust ground truth of information about the build process of an app to study the effects of different optimizations and obfuscations. We extended the open-source F-Droid build toolchain to create a dataset of more than one thousand apps, complete with their actual build configurations.

We found that bytecode-based approaches, such as SimiDroid, which depend on class, method, and variable names, do not work well in the face of R8or other code-based obfuscation. BinDiff performs comparable or better, which shows that applying binary tools to Android app analysis can bridge the gap between binary and bytecode analysis to take advantage of mature approaches and the advances made by both communities in the area of app diffing.

To further research in this area, we will provide all analysis artifacts, including the source code and benchmark dataset.

Acknowledgements

The research leading to these results has been funded by the Vienna Science and Technology Fund (WWTF) through project ICT19-056 and SBA Research. SBA Research (SBA-K1) is a COMET Centre within the framework of COMET Competence Centers for Excellent Technologies Programme and is funded by BMK, BMDW, and the federal state of Vienna. The COMET Programme is managed by FFG.

References

[1] S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo. “Obfuscapk: An Open-source Black-box Obfuscation tool for Android Apps”. In: *SoftwareX* 11 (2020). Source code: https://github.com/ClaudiuGeorgiu/Obfuscapk.

[2] A. Desnos. “Android: Static Analysis Using Similarity Distance”. In: *Proc. of the Hawaii International Conference on System Sciences (HICSS)*. 2012.

[3] F-Droid. *F-Droid Data Repository*. https://gitlab.com/fdroid/fdroiddata. 2021. (Visited on 11/30/2021).

[4] Facebook. *Redex: An Android Bytecode Optimizer*. https://fbredex.com.

[5] Google. *Configuring ART*. https://source.android.com/devices/tech/dalvik/configure. 2021.

[6] Google. *Shrink, obfuscate, and optimize your app*. https://developer.android.com/studio/build/shrink-code. 2021. (Visited on 10/08/2021).

[7] Guardsquare. *ProGuard: Shrink your Java and Android code*. https://github.com/Guardsquare/proguard. 2021.

[8] Hex-Rays. *IDA Pro*. https://hex-rays.com/ida-pro/.

[9] IKARUS Security Software GmbH. *Elsim*. https://github.com/IKARUSSoftwareSecurity/elsim. 2019. (Visited on 07/30/2021).

[10] J. Koret. *Diaphora*. http://diaphora.re. Source code: https://github.com/joxeankoret/diaphora. 2019. (Visited on 11/30/2021).

[11] L. Li, T. F. Bissyandé, and J. Klein. “SimiDroid: Identifying and Explaining Similarities in Android Apps”. In: *Proc. of the IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom)*. Source code: https://github.com/lilicoding/SimiDroid. 2017.

[12] NSA. *Ghidra*. https://ghidra-sre.org.

[13] Zynamics. *BinDiff*. https://www.zynamics.com/bindiff.html. 2021. (Visited on 06/25/2021).