Do you have 5 min? Improving Call Graph Analysis with Runtime Information

Jordan Samhi
University of Luxembourg
Luxembourg, Luxembourg
jordan.samhi@uni.lu

Marc Miltenberger
Fraunhofer SIT
Darmstadt, Germany
marc.miltenberger@sit.fraunhofer.de

Marco Alecci
University of Luxembourg
Luxembourg, Luxembourg
marco.alecci@uni.lu

Steven Arzt
Fraunhofer SIT
Darmstadt, Germany
steven.arzt@sit.fraunhofer.de

Tegawendé F. Bissyandé
University of Luxembourg
Luxembourg, Luxembourg
tegawende.bissyande@uni.lu

Jacques Klein
University of Luxembourg
Luxembourg, Luxembourg
jacques.klein@uni.lu

Abstract

Constructing precise and sound call graphs is fundamental for effective static analysis, yet it remains a significant challenge in today's software. Traditionally, researchers have developed sophisticated algorithms to address this issue, often resulting in increased computational costs. *But what if we could provide a simple, cost-effective way to improve call graphs drastically?*

This paper introduces a simple method to largely enhance static call graphs almost *for free*, i.e., with 5 min of dynamic analysis and low overhead. Our approach improves the soundness of call graphs, thereby benefiting any downstream static analyses based on call graphs, such as data flow analysis. We demonstrate the efficacy of our method on Android apps by integrating it with FlowDroid, the leading static analysis tool for Android apps. Additionally, we outline future directions for achieving even more accurate and sound call graphs in static analysis.

1 Introduction

Call graphs are fundamental structures in static analysis. They represent the calling relationships between procedures or methods within a program. They are primarily used to understand program behaviors in interprocedural contexts, e.g., with interprocedural data flow analyses. Over the years, researchers have put huge efforts into improving call graph construction techniques to handle the complexities of modern programming practices. For instance, techniques have been introduced to address challenges posed by reflection [1–5], dynamic loading, callbacks, inter-component communications [6–9], threading, native code integration [10, 11], etc. These mechanisms, while powerful for developers, introduce dynamic and implicit calling relationships that are difficult to capture accurately with traditional static analysis methods.

However, these advanced techniques come with significant computational costs. As the complexity of software increases, the resources required to calculate the sound call graphs increase drastically [12]. Despite these efforts, recent studies [12, 13], indicate that the soundness of call graphs is still of low quality, i.e., they miss many edges. For example, in [12], the authors recently showed that at least 40% of the methods are missed in call graphs built with the biggest overapproximation (i.e., CHA [14]). This is mainly due to the inherent difficulties in accounting for implicit calls, libraries, frameworks, constructor methods, or other language-specific features (e.g., reflection and dynamic loading). This leads to incomplete static analyses. Addressing these challenges not only requires significant research effort but also highlights the computational costs, making analyses impractical for large software systems.

This paper presents a radical new idea that offers a straightforward and efficient solution to improve call graph construction almost *for free*. Our approach is unique in that it requires minimal computational resources –only about *5 minutes*– making it practical and cost-effective and does not necessitate extensive computational power or complex algorithms.

Our approach is straightforward and consists of two main steps: ❶ *compute a dynamic call graph*: we generate a dynamic call graph by instrumenting a given app and collecting runtime data during execution; ❷ *enhance the static call graph*: we extract entry points from the dynamic call graph and incorporate them into the static call graph construction algorithm as additional entry points.

Hybrid analysis, i.e., the integration of dynamic and static analysis, is already known, well described by Ernst et al. [15]. While some techniques have adopted hybrid analysis, these efforts typically focus on specific use cases, such as vulnerability detection [16–19] (i.e., *at the analysis level*) or addressing limited mechanisms, such as reflection [20] (i.e., *at the model level*). In contrast, the novelty of our approach lies in leveraging dynamic analysis to identify *new entry points* for call graphs to improve the overall quality of the call graph in general, i.e., not specific mechanisms.

We demonstrate the efficiency of our novel idea through evaluation of real-world Android apps–highly event-driven software systems. Our results show significant improvements in the soundness of call graphs with only 5 min of dynamic analysis. The advancement brought by our technique not only benefits static analysis, but also has broader implications for software engineering practices that rely on program behavior (e.g., learning practices). Although our idea marks a step forward, we acknowledge the need to go further. Future work needs to focus on refining our method to address the remaining challenges posed by: ① dynamic analysis; and ② implicit program behaviors, i.e., connecting discontinuities.

Artifacts. We make all our artifacts available: https://anonymous.4open.science/r/Improving-Call-Graph-With-Runtime-Information-E6EA

The remainder of the paper is as follows. First, in Section 2, we describe the problem of call graph construction in general. Then, we provide our novel idea to improve call graphs in Section 3. In section 4, we propose a first proof of concept of our technique.

Section 5 describe our empirical setup, and Section 6 describe our preliminary results. Eventually, we provide future research directions in Section 7 and conclude in Section 8.

2 Problems with Call Graph Construction

Call graph construction algorithms require explicit entry points to start building a call graph. Traditional software has a clear starting point, i.e., a *main* method, from which a call graph can be built. However, modern software that is event-driven, such as mobile apps, web apps, or GUI programs, poses challenges to call graph construction algorithms. Indeed, these programs do not have single entry points, which hinders sound call graph construction. Let us explain: ❶ In event-drive software, the flow of execution is determined by events (e.g., user inputs such as clicks, swipes, etc., or sensors) rather than a sequential main program flow. There is no single entry point like a main method. Instead, multiple event handlers or callbacks serve as potential starting points for execution (e.g., when a user clicks on a button, this would trigger and start a new action that is not in the main execution flow); ❷ Many methods, particularly event handlers, are not explicitly called within software code. Instead, they are invoked by libraries, frameworks, or runtime environment in response to specific events. These implicit invocations mean that static analysis tools may miss these methods, as they do not appear in the software code under analysis; ❸ Event handler methods act as entry points in call graphs. However, since their invocation is managed externally, they are not visible in the software code. Hence, identifying these entry points requires understanding the interaction of software with libraries, frameworks, or the runtime environment, which is challenging because it is computationally expensive; ❹ Some of these mechanisms are already known to the community. However, due to the lack of comprehensive documentation, and with frequent updates and variations across versions, it is challenging to keep comprehensive lists of these mechanisms up to date.

Let us now consider an example of such a mechanism in Java with the Spring framework [21]. With Spring, methods can be invoked implicitly through annotations such as `@PostConstruct`. These methods are called directly by the framework during specific phases of the software, lifecycle, without any explicit invocation in the software code itself. Listing 1 depicts a code example of this mechanism.

```
1  @Component
2  public class ComplexService {
3      public ComplexService() { /* constructor */ }
4      @PostConstruct
5      public void init() { /* do something */ }
6  }
```

Listing 1: An example of an implicit mechanism with the Spring Java framework

In this example, the `@Component` annotation (line 1) tells Spring to detect this class and create a bean instance [21]. When the bean is instantiated, the constructor (line 3) is called. The `@PostConstruct` annotation (line 4) marks the `init()` method (line 5) to be called implicitly by the Spring framework. *There is no explicit call to* `init()` *in the software code, i.e., the developer never calls this method in its code.* This means the control flow leading to `init()` is **not visible in the software code**. Therefore, traditional call graph construction algorithms, which will not see a call to `init()`, will not include this method in the call graph and might miss a substantial amount of code, which makes the final call graph unsound and leads to *incomplete static analysis*.

3 Proposed Idea to Improve Call Graphs

Our idea lies in *dynamic analysis*. We aim to improve static call graphs with data collected during dynamic analysis. Dynamic analysis observes a software behavior and captures **precise** information, such as methods invocations, including those that are difficult to detect statically. Indeed, if an event handler, a callback, or any other implicit mechanism is called (which static analysis might miss), it is observed during a dynamic analysis.

Our approach is therefore straightforward: we aim to collect runtime data to construct a *dynamic call graph*, which represents the calling relationships between methods that happened at runtime, i.e., it is *highly precise*.

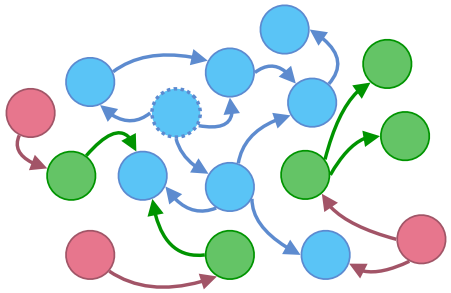

Figure 1: Example of how static call graph is improved. Blue nodes are static call graph nodes, the dotted node is the entry point. Red nodes are dynamic call graph entry points. Green nodes are new static nodes discovered thanks to the dynamically discovered entry points

Eventually, given a dynamic call graph, we can easily identify its entry points, i.e., the root nodes of this dynamic call graph. The set of entry points includes –provided they have been executed– implicit mechanisms since they were never called by the software itself but by, e.g., an external framework. As a result, improving static call graph construction with these entry points becomes straightforward since they can easily be integrated into the call graph construction algorithm as additional entry points, as shown in Figure 1: **this is the core of our novel idea**.

4 Proof of Concept

In this paper, we aim to demonstrate our novel idea with a first proof of concept. Our proof of concept focuses on the Android ecosystem as a use case given its highly event-driven nature. Nonetheless, the underlying concept is broadly applicable to any software system. This section describes how we achieved Android call graph improvement with our technique. We remind that our technique lies in two main steps: ① a dynamic analysis to build a dynamic

call graph; and ② the dynamic call graph is used to improve the static call graph.

4.1 Dynamic Analysis

To obtain the dynamic call graph of a given app, we used AppRunner [22], which instruments the target app to send call graph events. The host computer establishes a network connection with the app and receives the call graph events. AppRunner uses the call graph events to build a shadow stack for each thread, mimicking the Dalvik stack in the target app. Note that since AppRunner instruments the app code, these shadow stacks only contain app code, due to the Dalvik system code not being instrumentable. AppRunner utilizes the shadow stacks to build a dynamic call graph. Finally, we output the dynamic call graph as JSON files, with a separate entry for each edge. Each entry identifies the caller statement and the encompassing method as well as the callee method signature. We modified AppRunner to compute and output a set of "active components" for each entry. This set contains Android components that were active during that call. When the dynamic stack trace of the corresponding thread contains a callback method of an Android component, we consider this component active. Note that AppRunner links threads together, i.e., when UI thread calls `Thread.start`, the spawned thread will be linked to UI thread.

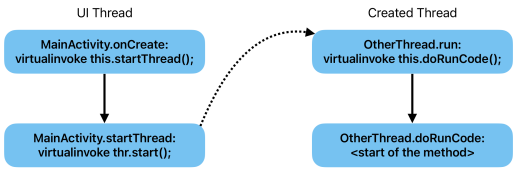

Figure 2: Stackframes of the UI thread and a spawned thread. AppRunner combines the threads using a thread edge.

As an example, consider Figure 2. Assume that we want to insert the edge from `OtherThread.run` to `OtherThread.doRunCode` into the dynamic callgraph. In order to determine the active component, we traverse the dynamic stack backward until we reach the entry point of the thread. Since the `OtherThread.run` method is not a component callback, we follow the thread edge created by AppRunner to the UI thread. When starting a thread, AppRunner creates a snapshot of the dynamic stack trace at the thread starting location, in this case, `virtualinvoke thr.start()` in the `startThread` method. We then follow the dynamic stack of the UI thread in the same fashion as the created thread. This time, we encounter the `MainActivity.onCreate` callback. Thus, `MainActivity` is the active component for this edge. Note that there could be other components that call `MainActivity.startThread` or create thread that runs `OtherThread.run`. In this case, these other components would also be considered active components, which is why we collect a *set* of active components. Note that this algorithm does not always lead to an active component. For example, when the user taps on a button, the UI thread has no application code stack trace prior to the `onClick` handler. In these cases, we consider the currently visible activity as the active component.

4.2 Static Analysis

In this work, we modified the state-of-the-art static analysis tool FlowDroid [23] with runtime execution data. We have modified FlowDroid to take runtime data into account by adding an engine that loads dynamic call graphs, which are provided as JSON files generated from the previous dynamic analysis phase. This engine not only parses the dynamic call graph data but also computes the entry points and establishes a mapping between each entry point and its corresponding Android component. For each entry point identified in the dynamic call graph, we applied the following strategy: ① For entry points that are attached to a component—such as an Activity or Service—we insert a call to the corresponding method within the dummy main method of that component (FlowDroid internally constructs these dummy main methods to simulate the lifecycle and callback methods of Android components); and ② For entry points not associated with any specific component, we add calls to these methods in the dummy main method of the application itself. After integrating the dynamic entry points, we execute FlowDroid's call graph construction algorithm, which relies on Soot [24] and the SPARK [25] algorithm (the default call graph construction algorithm for FlowDroid).

5 Empirical Setup

This section describes the setup of our experiments.

Dataset. For our experiments, we relied on a dataset of 100 apps from 2024 randomly collected from the AndroZoo repository [26].

Experiment Setup. We ran all of our experiments on a server machine with 144 Intel Xeon Gold 6154 CPU cores and 3 TB of physical memory. Note that we limited the memory for the used JVMs, which is described respectively for dynamic and static analysis below.

Dynamic Analysis. For each app of our dataset, we ran Monkey [27] on a Samsung XCover Pro with Android 13 for 5 min. We used AppRunner to collect the dynamic call graph data. AppRunner utilizes OpenJDK 22, and we specified a maximum heap size of 500 GiB. *Note:* our study does not aim to reach high code coverage during code execution, rather we show that simple dynamic analysis can lead to substantial static call graph improvement. Nevertheless, ① recent studies show that though more sophisticated approaches exist, Monkey still achieves the best coverage performace [28, 29]; and ② it has also recently be shown that after 5 min, the proportion of code covered using Monkey reaches a "plateau" [29].

Static Analysis. For each app of our dataset, we have run FlowDroid with and without the dynamic call graph improvement to compare the call graphs together. We used OpenJDK 19 and limited the maximum memory size to 50 GiB.

6 Preliminary Results

This section presents the results of our empirical study.

Apps successfully analyzed. First, we note that from the 100 random apps, we were able to extract 78 dynamic call graphs. 12 apps were split APKs, meaning the code and resources are distributed among multiple APK files. Currently, AppRunner does not support such split app yet. In one case, the app used the Jiagu packer. In 7 cases, the original app was broken prior to any modification. In two apps, we encountered an error with the Soot framework (which FlowDroid relies on).

Call graph comparison of 78 random apps. The average and median numbers of nodes in the call graph computed by FlowDroid without the dynamic call graph data are, respectively, 50 626 and 25 899. Concerning the call graph generated by FlowDroid with dynamic call graph data, the numbers are, respectively, 65 534 and 46 307. This result shows the improvement in the number of nodes covered thanks to the dynamic data, i.e., up to 15 000 more nodes on average, as shown by Table 1. We remind that the nodes added as entry points to the static call graph are of the *highest degree of precision* since they were collected dynamically.

Table 1: Comparison of call graph size with and without runtime data (RD = Runtime Data, RT = Run Time)

	Average # of nodes	Median # of nodes	Average RT	Median RT
Without RD	50 626	25 899	367s	123s
With RD	65 534	46 307	444s	189s

Figure 3 shows the distribution of the number of nodes in the call graphs generated by FlowDroid with and without runtime data on the 100 random apps.

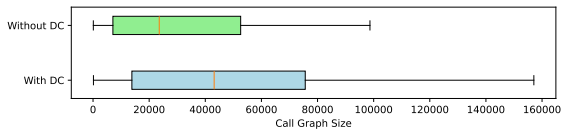

Figure 3: Distribution of the number of nodes in the call graphs generated by FlowDroid with and without runtime data on the 100 random apps (DC = Dynamic Call Graph)

Overhead. But, does our technique bring too much static computation overhead? We have measured the time taken to improve call graphs for each app with and without the runtime data. The distribution of the time taken to compute call graphs is shown in Figure 4. The average time taken without runtime data is 367 seconds, and this increases to 444 seconds when runtime data is included. The median times are 123 seconds without runtime data and 189 seconds with it. This indicates that on average, only an additional 77 seconds are needed to improve and augment the call graph by up to 15 000 nodes through the static analysis, i.e., this does not include the 5 minutes required for dynamic analysis.

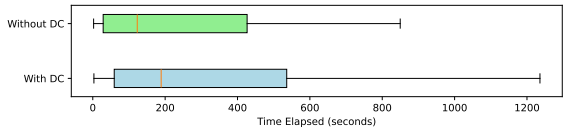

Figure 4: Distribution of the number of seconds needed to run the call graph construction algorithm on the 100 random apps. (DC = Dynamic Call Graph)

Data Flow Analysis. Additionally, we have run FlowDroid with the default configuration and the default list of sources and sinks on the 78 apps without and with the runtime data. Without the runtime data, FlowDroid could find 105 data flows, whereas with the runtime data it could find 152 data flows. These results indicate that improving call graphs also improves downstream analysis, which can then analyze more relevant code that was previously overlooked (soundness improved at the model level). We manually checked 10 data flows that were not reported without the runtime data and confirm that they are true positive, i.e., there is a path in the code from the source to the sink.

7 Future Plans

This section outlines the work we plan to do to turn our first proof of concept into a better and more sophisticated approach. Our plan spans over 8 major research directions:

(1) We first plan to conduct larger experiments, i.e., with more apps to better highlight the improvement at a large scale.
(2) We also intend to measure the improvement in terms of–not only call graph–data flow improvement, i.e., can our technique provide more and better data flows (to, e.g., detect data leaks).
(3) We plan to investigate the quality of the edges that are added in the static call graph thanks to the runtime data, i.e., are they true-positives or false-positives?
(4) We also plan to improve FlowDroid by developing an all-in-one solution that integrates a dynamic analysis phase—such as a five-minute execution—and automatically incorporates the resulting dynamic call graph into the static analysis.
(5) We intend to explore methods for better linking between components at the method level. For instance, an entry point collected from dynamic analysis that originates from an external framework might be connected to another method, from which it was called. For instance, the connection between the `start()` and `run()` methods (from the `Thread` class), where `run()` is considered an entry point in the dynamic call graph since it is triggered by Java itself and not in the client code. To enhance data flow analysis, we need to find solutions that accurately represent these method connections within the call graph.
(6) We plan to conduct an analysis by merging the static and dynamic call graphs directly instead of only considering entry points and let the call graph construction algorithm to build the graph. Then we would compare the results with those obtained from our current proof of concept.
(7) We aim to compare our methodology using different call graph construction algorithms. Indeed, our current approach relies on SPARK, we are interested in evaluating other algorithms such as CHA, RTA, or VTA.
(8) We plan to generalize our approach to other programming languages and environments, such as Java EE.

8 Conclusion

In this paper, we have presented a radical new idea offering a straightforward and cost-effective solution to drastically improve the soundness of call graphs. Our preliminary results indicate that this technique improves call graphs and can be further refined to offer even better call graphs to static program analysis tools.

9 Data Availability

To promote transparency and facilitate reproducibility, our artifacts are publicly available: https://anonymous.4open.science/r/Improving-Call-Graph-With-Runtime-Information-E6EA

10 Acknowledgements

This research was funded in whole, or in part, by the Luxembourg National Research Fund (FNR), grant references 16344458 (REPROCESS) and 18154263 (UNLOCK). This research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 318–329. [Online]. Available: https://doi.org/10.1145/2931037.2931044

[2] X. Sun, L. Li, T. F. Bissyandé, J. Klein, D. Octeau, and J. Grundy, "Taming reflection: An essential step toward whole-program analysis of android apps," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, p. 36, 2021.

[3] J. Gajrani, U. Agarwal, V. Laxmi, B. Bezawada, M. S. Gaur, M. Tripathi, and A. Zemmari, "Espydroid+: Precise reflection analysis of android apps," *Computers & Security*, vol. 90, p. 101688, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404819302251

[4] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Reflection-aware static analysis of android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 756–761. [Online]. Available: https://doi.org/10.1145/2970276.2970277

[5] Z. Cheng, F. Zeng, X. Zhong, M. Zhou, C. Lv, and S. Guo, "Resolving reflection methods in android applications," in *2017 IEEE International Conference on Intelligence and Security Informatics (ISI)*, 2017, pp. 143–145.

[6] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1329–1341. [Online]. Available: https://doi.org/10.1145/2660267.2660357

[7] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 280–291.

[8] J. Samhi, A. Bartel, T. F. Bissyande, and J. Klein, "Raicc: Revealing atypical inter-component communication in android apps," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2021, pp. 1398–1409. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00126

[9] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*, vol. 15, 2015, p. 110.

[10] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein, "Jucify: A step towards android code unification for enhanced static analysis," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 1232–1244. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3510003.3512766

[11] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1137–1150. [Online]. Available: https://doi.org/10.1145/3243734.3243835

[12] J. Samhi, R. Just, T. F. Bissyandé, M. D. Ernst, and J. Klein, "Call graph soundness in android static analysis," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 945–957. [Online]. Available: https://doi.org/10.1145/3650212.3680333

[13] D. Helm, S. Keidel, A. Kampkötter, J. Düsing, T. Roth, B. Hermann, and M. Mezini, "Total recall? how good are static call graphs really?" ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 112–123. [Online]. Available: https://doi.org/10.1145/3650212.3652114

[14] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, M. Tokoro and R. Pareschi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 77–101.

[15] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," pp. 24–27.

[16] D. Chaulagain, P. Poudel, P. Pathak, S. Roy, D. Caragea, G. Liu, and X. Ou, "Hybrid analysis of android apps for security vetting using deep learning," in *2020 IEEE Conference on Communications and Network Security (CNS)*, 2020, pp. 1–9.

[17] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. v. d. Veen, and C. Platzer, "Andrubis – 1,000,000 apps later: A view on current android malware behaviors," in *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014, pp. 3–17.

[18] X. Wang, Y. Yang, and S. Zhu, "Automated hybrid analysis of android malware through augmenting fuzzing with forced execution," *IEEE Transactions on Mobile Computing*, vol. 18, no. 12, pp. 2768–2782, 2019.

[19] S. Rasheed and J. Dietrich, "A hybrid analysis to detect java serialisation vulnerabilities," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 1209–1213. [Online]. Available: https://doi.org/10.1145/3324884.3418931

[20] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 241–250. [Online]. Available: https://doi.org/10.1145/1985793.1985827

[21] VMWare, "Java spring framework, https://spring.io," 2024, accessed October 2024.

[22] M. Miltenberger and S. Arzt, "Extensible and scalable architecture for hybrid analysis," in *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, ser. SOAP 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 34–39. [Online]. Available: https://doi.org/10.1145/3589250.3596146

[23] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM SIGPLAN NOTICES*, vol. 49, no. 6, p. 259–269, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594299

[24] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, p. 13.

[25] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," in *Proceedings of the 12th International Conference on Compiler Construction*, ser. CC'03. Berlin, Heidelberg: Springer-Verlag, 2003, p. 153–169.

[26] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[27] Google, "Android monkey, https://developer.android.com/studio/test/monkey," 2023, accessed July 2023.

[28] W. Wang, W. Lam, and T. Xie, "An infrastructure approach to improving effectiveness of android ui testing tools," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 165–176. [Online]. Available: https://doi.org/10.1145/3460319.3464828

[29] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An empirical study of android test generation tools in industrial cases," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 738–748.