A Journey Through Android App Analysis: Solutions and Open Challenges

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

ABSTRACT

Users can today download a wide variety of apps ranging from simple toy games to sophisticated business-critical apps. They rely on these apps daily to perform diverse tasks, some of them related to sensitive information such as their finance or health. Ensuring high-quality, reliable, and secure apps is thus key. In the TruX research group of the interdisciplinary center for Security, Reliability, and Trust (SnT) of the University of Luxembourg, we are working for about 10 years to deliver practical techniques, tools, and other artifacts (such as repositories) making the analysis of Android apps possible. In this paper, we will briefly introduce our key contributions in both (1) Android app static analysis to detect security issues, and (2) Android Malware Detection with machine learning. We will conclude by listing several open challenges that we are currently facing towards improving the analysis and security of Android apps.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → **Software verification and validation**.

KEYWORDS

Static Analysis, Malware Detection, Android Security, Software Security

ACM Reference Format:
Jacques Klein. 2021. A Journey Through Android App Analysis: Solutions and Open Challenges. In *Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems (ASSS '21), June 7, 2021, Virtual Event, Hong Kong*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3457340.3458298

1 INTRODUCTION

We are in 2021 and Android is a popular teenager. "Teenager" because the first smartphone running on Android was released in 2008. A few weeks ago, I discussed with my 15 year old daughter who is digital native and she was doubtful when I claimed that she is older than the first Android smartphone. "Popular" because the user base of Android constantly grew to reach over 2 billion monthly active users. With 3 million apps on Google Play, the official Android app store, the number of available apps is also impressive. In the meantime, the research community dedicated considerable effort in inspecting and analyzing both the Android framework and the Android apps. A simple query on Google Scholar returns more than 1 million entries for the word "Android". The appetite of the researcher community is not only explained by the marketing success of Android. The Android framework is open-source and even if the apps are released as apk files, unpacking the apks is possible making most of the code accessible and ready to be analyzed.

In this paper, we present our contribution towards analyzing Android apps. More specifically, in Section 3, we first explain how we pioneered the domain of Android app static analysis. Then, we present our contribution towards Android app modeling and sensitive/private data flow analysis. We also detail several studies that we conducted by leveraging Android app "lineages". Finally, we list several ad hoc static analyses we performed to check diverse app properties. In Section 4, we present our works towards malware detection, mostly by relying on machine-learning based approaches. We first present AndroZoo, a large repository of Android apps that we made available to the research community. Then, we explain several validation protocol biases that can occur when assessing ML based malware detectors. Finally, we present our contributions related to the detection of both repackaged and piggybacked apps. In Section 5, we list several open challenges related to both static analysis and malware detection.

2 ACKNOWLEDGMENT

The work presented in this paper is not only my "own" work. I am not even first author of a single publication cited in this paper. So, I would like to gratefully thank all my co-authors[1] and colleagues for their support and fruitful collaborations. I have plenty of great stories to tell about these collaborations. As researchers, we are privileged to travel, discover new places, meet, discuss and exchange with great people. In addition, all these achievements would not have been possible without the support of a strong research team. In this regard, I was a member of the Serval SnT team leaded by Prof. Le Traon, and since 2019, I have the privilege to co-head the TruX team together with my colleague Prof. Bissyandé. Big thanks to both of them. I also thank all the PhD students that I have supervized or I am supervizing on topics related to Android app analysis: Dr. A. Bartel, Dr. K. Allix, Dr. L. Li, Dr. M Hurier, Dr. J. Gao, Dr. P. Kong, Mr. T. Riom, Ms. N. Doudi, Mr. J. Samhi.

[1]Thanks to D. Octeau, M. Monperrus, P. McDaniel, S. Rasthofer, E. Bodden, H. Cai, S. Arzt, T. Liu, F. Dong, Y. Guo, D. Lo, Y. Zhao, L. Ma, X. Xia, L. Cavallaro, S. Jha, C. Fritz, Y. Zhao, J. Grundy, Z. Liu, X. Luo, X. Sun, etc., but sorry, I cannot cite all

I would also like to thank Prof. Weizhi Meng and Prof. Li Li, the organizers of the ASSS workshop (https://asss.compute.dtu.dk/2021/), for having inviting me to present a keynote speech at the ASSS workshop.

Finally, please note that this work has been partially supported (1) by the Luxembourg National Research Fund (FNR), under projects CHARACTERIZE C17/IS/11693861, (2) and by the SPARTA project, which has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 830892.

3 OUR CONTRIBUTIONS TOWARDS STATICALLY ANALYZING ANDROID APPS

With my colleagues, we developed and publicly released tools making the static analysis of Android apps possible. Below we list some of our contributions related to this topic (this list is not exhaustive).

3.1 Making the app analyzable

In early 2000, when the first Android apps were uploaded into Google Play, it was not possible to directly use static analyzers such as Soot [33] or Wala [10] to analyze Android apps. Indeed, while Android apps are mostly developed in Java (or Kotlin), the resulting executable files are released as ".dex" files that contain Dalvik bytecode. This bytecode differs from traditional Java bytecode. Several researchers worked on this issue and proposed tools to translate Dalvik bytecode into another representation that can be used by an already existing static analyzer. We contributed to this effort by developing Dexpler [8], a module integrated today in Soot, that converts Dalvik bytecode into Jimple, the Soot intermediate representation. To sum up, we made Soot able to process Android apps.

3.2 Android app Modeling for Sensitive Data Flow Detection

Being now able to analyze Android apps, we started to search for security issues such as sensitive data flow leakages (also called privacy leaks). In 2012/2013, we started a fantastic collaboration with colleagues from Penn State University in the USA and TU Darmstadt in Germany. This collaboration yielded popular state-of-the-art tools such as FlowDroid [7] (presented at PLDI 2014), EPICC [30] (presented at Usenix Security 2013, and further extended by [29] and [28]), and finally ICCTA [18] (presented at ICSE 2015). These tools have been designed to detect security issues in Android apps. For instance, FlowDroid can be used to detect data leaks in apps. It was probably the first approach proposing to model the specificities of Android apps (lifecycle, callback methods, ...) making a sound analysis possible.

Android apps are made of components that communicate between each others via Inter-Component Communication (ICC) methods such as `startActivity`, which takes an `Intent` object as parameter. ICCTA relies on EPICC to retrieve information from the Intent object and then computes the ICC links, i.e., the links between components (e.g., Activity A communicates with Activity B). Thanks to this, ICCTA was able to extend FlowDroid by supporting inter-component analysis (including inter-app analysis as shown in [19]). However, typical static analysis limitations remain. In particular, if an ICC method such `startActivity` is called by reflection, ICCTA fails to catch this method call. In 2016, we propose the DroidRA [21] instrumentation-based approach to address this issue in a non-invasive way. With DroidRA, we reduce the resolution of reflective calls to a composite constant propagation problem. We leverage the COAL solver [29] to infer the values of reflection targets and app, and we eventually instrument this app to include the corresponding traditional Java call for each reflective call. Our approach allows to boost an app so that it can be immediately analyzable by ICCTA (and more generally static analyzers that were not reflection-aware). Recently, in [32] we extended DroidRA to, for instance, also consider Fragment.

Our last work on the topic of private data flow detection will be presented at ICSE 2021 [31]. In this work, we show that tools such as FlowDroid+ICCTA, Amandroid [34], or EPICC do not model all existing ICC methods. Indeed, in addition to usual ICC methods such as `startActivity`, the framework provides other atypical ways of performing ICCs. To address this limitation in the state of the art, we propose RAICC [31] a static approach for modeling new ICC links and thus boosting previous analysis tasks such as ICC vulnerability detection, privacy leaks detection, malware detection, etc.

3.3 App lineages to perform evolutionary studies

Software evolution is a key topic in software engineering and software security. Like other software artifacts, Android apps evolve. In the literature, the set of the successive versions of a given app is defined as "app lineage". However, investigating these app lineages, i.e., the evolution of Android apps, is not trivial. Indeed, Android developers update their apps by providing new apk files, and these apks have to be published via relevant markets. Nevertheless, mainstream Android app markets including the official market Google Play provide apps as a fleeing data stream where only the latest version of an app is available: when the next updated version is uploaded, the past version is lost. This causes one of the main difficulties to re-construct the lineage of Android apps.

In [13], we explained how we re-construct the versioned lineages of Android apps, by leveraging AndroZoo [5], a popular Android application repository made available to researchers. Then, we performed a large-scale investigation on how vulnerabilities evolve in Android apps. We fully rely on static vulnerability detection tools (FlowDroid [7], AndroBugs [27], and IC3 [29]) and report their results on consecutive versions of Android apps. We investigate specifically 10 vulnerability types associated with 4 different categories related to common security features (e.g., SSL), its sandbox mechanism (e.g., Permission issues), code injection (e.g., WebView RCE vulnerability) as well as its inter-app message passing (e.g., Intent spoofing). Among the main findings, we can cite that: (1) Most vulnerabilities will survive at least 3 updates. (2) Some third-party libraries are major contributors to most vulnerabilities detected by static tools. (3) Some vulnerabilities reported by detection tools may foreshadow malware.

In an MSR paper [11], we presented our attempt to learn crypto-APIs usage from the crowd, i.e., by mining crypto-APIs usage rules from app lineages. Android app developers recurrently use crypto-APIs to provide data security to app users. Unfortunately, misuse of APIs only creates an illusion of security and even exposes apps to systematic attacks. It is thus necessary to provide developers with a statically enforceable list of specifications of crypto-API usage rules. On the one hand, such rules cannot be manually written as the process does not scale to all available APIs. On the other hand, a classical mining approach based on typical usage patterns is not relevant in Android, given that a large share of usages include mistakes. In [11], building on the assumption that "developers update API usage instances to fix misuses", we proposed to mine the app lineages dataset to infer API usage rules. Eventually, our investigations yield negative results on our assumption that API usage updates tend to correct misuses. Actually, it appears that updates that fix misuses may be unintentional: subsequent updates quickly re-introduce the same misuses patterns.

3.4 Other static analyses performed on Android apps

Besides data flow analysis and studies on app lineages, we investigated other properties of Android apps by leveraging static analysis tools. We present in the following some of these studies:

Common Libraries: By mining 1.5 million apps from Google Play, we released to the community a list of 1,113 common libraries (including 240 libraries for advertisement) used in Android apps [23] (further extended by a journal paper [26]). This list can be used to identify the code that has been actually written by the developer of a given Android app.

Direct Inter-app Code Invocation: The Android ecosystem offers different facilities to enable communication among app components and across apps to ensure that rich services can be composed through functionality reuse. At the heart of this system is the Inter-component communication (ICC) scheme, which has been largely studied in the literature. Less known in the community is another powerful mechanism that allows for direct inter-app code invocation which opens up for different reuse scenarios, both legitimate or malicious. In a FSE 2020 paper [12], we exposed the general workflow for this mechanism, which beyond ICCs, enables app developers to access and invoke functionalities (either entire Java classes, methods or object fields) implemented in other apps using official Android APIs. We experimentally showcased how this reuse mechanism can be leveraged to "plagiarize" supposedly protected functionalities. Typically, we could leverage this mechanism to bypass security guards that a popular video broadcaster has placed for preventing access to its video database from outside its provided app. We further contributed with a static analysis toolkit, named DICIDer, for detecting direct inter-app code invocations in apps.

Compatibility Issues: The Android API provides the necessary building blocks for app developers to harness the functionalities of the Android devices, including for interacting with services and accessing hardware. This API thus evolves rapidly to meet new requirements for security, performance, and advanced features, creating a race for developers to update apps. Unfortunately, given the extent of the API and the lack of automated alerts on important changes, Android apps are suffered from API-related compatibility issues. These issues can manifest themselves as runtime crashes creating a poor user experience. In an ISSTA 2018 paper [22], we presented an automated approach named CiD for systematically modeling the lifecycle of the Android APIs and analyzing app bytecode to flag usages that can lead to potential compatibility issues. We demonstrated the usefulness of CiD by helping developers repair their apps, and we validated that our tool outperforms the state-of-the-art on benchmark apps that take into account several challenges for automatic detection.

Deprecated Android APIs: Because of functionality evolution, or security and performance-related changes, some APIs eventually become unnecessary in a software system and thus need to be cleaned to ensure proper maintainability. Those APIs are typically marked first as deprecated APIs and, as recommended, follow through a deprecated-replace-remove cycle, giving an opportunity to client application developers to smoothly adapt their code in next updates. Such a mechanism is adopted in the Android framework development where thousands of reusable APIs are made available to Android app developers. In a 2018 MSR paper [24] (further extended with a journal version [25]), we presented a research-based prototype tool called CDA and apply it to different revisions (i.e., releases or tags) of the Android framework code for characterizing deprecated APIs. Based on the data mined by CDA, we then performed an empirical study on API deprecation in the Android ecosystem and the associated challenges for maintaining quality apps. In particular, we investigated the prevalence of deprecated APIs, their annotations and documentation, their removal and consequences, their replacement messages, developer reactions to API deprecation, as well as the evolution of the usage of deprecated APIs. Experimental results reveal several findings that further provide promising insights related to deprecated Android APIs. Notably, by mining the source code of the Android framework base, we have identified three bugs related to deprecated APIs. These bugs have been quickly assigned and positively appreciated by the framework maintainers, who claim that these issues will be updated in future releases.

Mining Android Crash Fixes: Android apps are prone to crash. This often arises from the misuse of Android framework APIs, making it harder to debug since official Android documentation does not discuss thoroughly potential exceptions. Recently, the program repair community has also started to investigate the possibility to fix crashes automatically. Current results, however, apply to limited example cases. In both scenarios of repair, the main issue is the need for more example data to drive the fix processes due to the high cost in time and effort needed to collect and identify fix examples. In an ISSTA 2019 paper [16], we proposed a scalable approach, CraftDroid, to mine crash fixes by leveraging a set of 28 thousand carefully reconstructed app lineages from app markets, without the need for the app source code or issue reports. We developed a replicative testing approach that locates fixes among app versions that output different runtime logs with the exact same test inputs. Overall, we have mined 104 relevant crash fixes, further abstracted 17 fine-grained fix templates that are demonstrated to be effective for patching crashed apks. Finally, we release ReCBench, a benchmark consisting of 200 crashed apks and the crash replication scripts, which the community can explore for evaluating generated crash-inducing bug patches.

4 OUR CONTRIBUTIONS TOWARDS ANDROID MALWARE DETECTION

Malware is a real threat to the Android Ecosystem and its user base. Several studies have shown that malicious apps (i.e., malware) are present, in no low number, in Android markets including Google Play [6]. Researchers have proposed various and numerous approaches to detect Android malware. One of the most popular techniques used by researchers is the one based on machine-learning where the idea is to learn what is goodware, what is malware, and what discriminates malware from goodware. This learning is most of the time supervised, i.e., the training step is performed on a ground truth dataset, i.e., a dataset with well-labeled samples. In several related-works, the reported performance scores are extremely high when assessed with an "in the lab" setting. In Section 4.2, we report on several studies that we conducted to check if this high performance in the lab can translate into high performance in the wild. Before reporting our studies, in Section 4.1, we present AndroZoo, a large repository of Android apps that we built and shared with the research community. We also present our attempt to better label Android malicious samples. Finally, in Section 4.3, we briefly present our work aiming at detecting repackaged and piggybacked apps.

4.1 Large Datasets and Ground Truth: AndroZoo

Together with my colleagues, we are maintaining AndroZoo [5][2], a growing collection of Android Applications collected from several sources, including the official Google Play app market. Our AndroZoo repository currently contains more than three million apps, each of which has been analyzed by tens of different Antivirus products to know which apps are detected as Malware. We provide this dataset to contribute to ongoing research efforts, as well as to enable new potential research topics on Android Apps. By releasing AndroZoo to the research community, we also aim at encouraging our fellow researchers to engage in reproducible experiments. By March 2021, we have distributed more than 1100 AndroZoo access keys to researchers.

Having access to a large number of samples opens new research directions aiming at efficiently vetting apps, but reliable malware labels are a necessary input to guarantee the quality of both malware detection and classification models. Malware labeling, however, is not a trivial task. Manual labeling, where a human analyst inspects the actions of the malware in a bid to classify them, is prohibitively expensive, given the number of malware samples discovered every day. In such a setting, it is reasonable to rely on the collective judgment of Anti-Virus (AV) vendors who specialize in malware labeling. However, deriving a unified label from labels attached to samples by AV vendors is difficult. Inconsistencies in Anti-Virus (AV) labels are indeed common. This is due to both naming disagreements [14] across vendors and also a lack of adopted standards for naming malware. In particular, on the one hand, samples are often mis-labeled as different parties use distinct naming schemes for the same sample. On the other hand, samples are frequently misclassified due to conceptual errors made during labeling processes. In [15], we analyze the associations between all labels given by different vendors and we propose a system called EUPHONY to systematically unify common samples into family groups. The key novelty of our approach is that no a-priori knowledge on malware families is needed.

[2]The main maintainer is Dr. Kevin Allix

4.2 ML Assessment Protocols

To address the issue of malware detection through large sets of apps, researchers have investigated the capabilities of machine-learning techniques for proposing effective approaches. So far, several promising results were recorded in the literature, many approaches being assessed with what we call in the lab validation scenarios. In [3] (and previously in a short paper [2]), we revisited the purpose of malware detection to discuss whether such in the lab validation scenarios provide reliable indications on the performance of malware detectors in real-world settings, aka in the wild. To this end, we have devised several Machine Learning classifiers that rely on a set of features built from applications' CFGs. We use a sizeable dataset of over 50 000 Android applications collected from sources where state-of-the-art approaches have selected their data. We show that, in the lab, our approach outperforms existing machine learning-based approaches. However, this high performance does not translate into high performance in the wild. The performance gap we observed—F-measures dropping from over 0.9 in the lab to below 0.1 in the wild—raises one important question: How do state-of-the-art approaches perform in the wild?

In another paper [4], we consider the relevance of timeline in the construction of datasets, to highlight its impact on the performance of a machine learning-based malware detection scheme. Typically, we show that simply picking a random set of known malware to train a malware detector, as it is done in many assessment scenarios from the literature, yields significantly biased results. In the process of assessing the extent of this impact through various experiments, we were also able to confirm a number of intuitive assumptions about Android malware. For instance, we discuss the existence of Android malware lineages and how they could impact the performance of malware detection in the wild.

Recently, in [35], we reported another potential bias in machine-learning based malware detection. Datasets may include a large portion of duplicated samples, which indeed may bias recorded experimental results and insights. We performed extensive experiments to measure the performance gap that occurs when datasets are de-duplicated. Our experimental results reveal that duplication in published datasets has a limited impact on supervised malware classification models. This observation contrasts with the finding of Allamanis [1] on the general case of machine learning bias for big code. Our experiments, however, show that sample duplication more substantially affects unsupervised learning models (e.g. malware family clustering). Nevertheless, we argue that our fellow researchers and practitioners should always take sample duplication into consideration when performing machine learning (via either supervised or unsupervised learning) based Android malware detection, no matter how significant the impact might be.

4.3 Repackaging and Piggybacking Detection

We investigated two closely related activities that are still an issue for the Android ecosystem: Repackaging and Piggybacking

Repackaging is indeed a serious threat to the Android ecosystem as it deprives app developers of their benefits, contributes to spreading malware on users' devices, and increases the workload of market maintainers. In the space of six years (2012-2017), the research around this specific issue has produced 57 approaches that do not readily scale to millions of apps or are only evaluated on private datasets without, in general, tool support available to the community. In [20], through a systematic literature review of the subject, we argued that the research was slowing down, where many state-of-the-art approaches have reported high-performance rates on closed datasets, which are unfortunately difficult to replicate and to compare against. In [20], we proposed to reboot the research in repackaged app detection by providing a literature review that summarises the challenges and current solutions for detecting repackaged apps and by providing a large dataset (named RePack) that supports replications of existing solutions and implications of new research directions. As a baseline, we proposed a straightforward machine-learning based repackaging detection approach that yields reasonable performance scores.

Piggybacking: The Android packaging model offers ample opportunities for malware writers to piggyback malicious code in popular apps, which can then be easily spread to a large user base. Although researchers have proposed approaches and tools to identify piggybacked apps, the literature was lacking a comprehensive investigation into such a phenomenon. In [17], we filled this gap by 1) systematically building a large set of piggybacked and benign apps pairs, which we release to the community, 2) empirically studying the characteristics of malicious piggybacked apps in comparison with their benign counterparts, and 3) providing insights on piggybacking processes. Among several findings providing insights analysis techniques should build upon to improve the overall detection and classification accuracy of piggybacked apps, we showed that piggybacking operations not only concern app code, but also extensively manipulates app resource files, largely contradicting common beliefs. We also found that piggybacking is done with little sophistication, in many cases automatically, and often via library code.

5 OPEN CHALLENGES

We identified several challenges related to the static analysis of mobile apps.

(1) First, a common limitation of static analyzers, not specific to Android apps, is the potentially high number of false alarms. This is still an open challenge, and approaches allowing to automatically check that an alarm triggered by a static analyzer is an actual issue would be great.
(2) A second open challenge is related to the modeling of Android apps. Better models lead to more precise analyses. Related work such as FlowDroid [7] has proposed modeling of apps that allow overcoming specifies of Android apps (examples of such specificities are call back methods that can be triggered at any time by the Android system). However, recent works (e.g., [31]) have shown that this model can be incomplete. More works towards better modeling Android apps are still required.
(3) Binaries files (code written with the C language and then compile in *.so* files) are often not considered by static analyzers that usually focus on the Dalvik bytecodes (*.dex* files). Not analyzing binary files is currently an important threat to validity regarding the completeness of the static analyses.

We also identified several challenges related to the detection of Android malware by using AI techniques (mostly machine learning).

(1) Automatically locating malicious payloads and understanding what is the behavior of malware is definitely a big concern for the research community. Actually, as a starting point, it would be great to understand what really captures a machine-learning based detector.
(2) Second, a common challenge in the AI domain is representation learning. This challenge still holds for Android malware detection. Indeed, what is the best representation of Android apps to detect malware is still not an answered question.
(3) In addition to locating the malicious part(s) of an app, explaining the decisions of machine-learning based malware detectors, i.e., why an app has been classified as malware is still an open challenge. In the same direction, characterizing malware, and more specifically characterizing families of malware would be beneficial for the entire community.
(4) Finally, in a recent paper [9], we attempt a complete Reproduction of five Android Malware Detectors from the literature and discuss to what extent they are "Reproducible". Notably, we provided insights on the implications around the guesswork that may be required to finalize a working implementation. We also discussed how barriers to Reproduction could be lifted, and how the malware detection field would benefit from stronger reproducibility standards—like many various fields already have. So, a last open challenge that we could report is to improve artifacts (datasets, benchmarks, tools, etc.) availability and reproducibility in order to benefit the research community.

6 BIOGRAPHY OF THE AUTHOR

Prof. Dr. Jacques Klein is a researcher and professor in software engineering and software security who develops innovative approaches and tools towards helping the research and practice communities build trustworthy software. He is a member of the Interdisciplinary Centre for Security, Reliability and Trust (SnT) at the University of Luxembourg. He received a Ph.D. degree in Computer Science from the University of Rennes, France, in 2006. His main areas of expertise are threefold: (1) Software Security (Malware detection, prevention and dissection, Static Analysis for Security, Vulnerability Detection, etc.); (2) Software Reliability (Software Testing, Semi-Automated and Fully-Automated Program Repair, etc.); (3) Data Analytics (Multi-objective reasoning and optimization, Model-driven data analytic, Time Series Pattern Recognition, etc.). In addition to academic achievements, Prof. Klein also has a long-standing experience and expertise in successfully running industrial projects with several industrial partners in various domains by applying data analytics, software engineering, information retrieval, etc., to their research problems.

REFERENCES

[1] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 143–153.

[2] Kevin Allix, Tegawendé F. Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. 2014. Large-Scale Machine Learning-Based Malware Detection: Confronting the "10-Fold Cross Validation" Scheme with Reality. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy* (San Antonio, Texas, USA) *(CODASPY '14)*. Association for Computing Machinery, New York, NY, USA, 163–166. https://doi.org/10.1145/2557547.2557587

[3] Kevin Allix, Tegawendé F. Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. 2016. Empirical assessment of machine learning-based malware detectors for Android - Measuring the gap between in-the-lab and in-the-wild validation scenarios. *Empirical Software Engineering* 21, 1 (2016), 183–211. https://doi.org/10.1007/s10664-014-9352-6

[4] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Are Your Training Datasets Yet Relevant?. In *Engineering Secure Software and Systems. ESSoS 2015 (Lecture Notes in Computer Science, Vol. 8978)*. Springer, 51–67. https://doi.org/10.1007/978-3-319-15618-7_5

[5] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[6] K. Allix, Q. Jerome, T. F. Bissyandé, J. Klein, R. State, and Y. L. Traon. 2014. A Forensic Analysis of Android Malware – How is Malware Written and How it Could Be Detected?. In *2014 IEEE 38th Annual Computer Software and Applications Conference*. 384–393. https://doi.org/10.1109/COMPSAC.2014.61

[7] 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. *SIGPLAN Not.* 49, 6 (June 2014), 259–269. https://doi.org/10.1145/2666356.2594299

[8] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. 27–38.

[9] Nadia Daoudi, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. 2021. Lessons Learnt on Reproducibility in Machine Learning Based Android Malware Detection. *Empirical Software Engineering, accepted for publication on Feb. 26, 2021* (2021).

[10] Stephen J. Fink et al. [n.d.]. T.J. Watson Libraries for Analysis (WALA), http://wala.sourceforge.net/.

[11] J. Gao, P. Kong, L. Li, T. F. Bissyandé, and J. Klein. 2019. Negative Results on Mining Crypto-API Usage Rules in Android Apps. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 388–398. https://doi.org/10.1109/MSR.2019.00065

[12] Jun Gao, Li Li, Pingfan Kong, Tegawendé F. Bissyandé, and Jacques Klein. 2020. Borrowing Your Enemy's Arrows: The Case of Code Reuse in Android via Direct Inter-App Code Invocation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 939–951. https://doi.org/10.1145/3368089.3409745

[13] J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein. 2021. Understanding the Evolution of Android App Vulnerabilities. *IEEE Transactions on Reliability* 70, 1 (2021), 212–230. https://doi.org/10.1109/TR.2019.2956690

[14] Médéric Hurier, Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. On the Lack of Consensus in Anti-Virus Decisions: Metrics and Insights on Building Ground Truths of Android Malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.). Springer International Publishing, Cham, 142–162.

[15] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. Le Traon, J. Klein, and L. Cavallaro. 2017. Euphony: Harmonious Unification of Cacophonous Anti-Virus Vendor Labels for Android Malware. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 425–435. https://doi.org/10.1109/MSR.2017.57

[16] Pingfan Kong, Li Li, Jun Gao, Tegawendé F. Bissyandé, and Jacques Klein. 2019. Mining Android Crash Fixes in the Absence of Issue- and Change-Tracking Systems. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 78–89. https://doi.org/10.1145/3293882.3330572

[17] Li, Daoyuan Li, Tegawende F. Bissyande, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. 2017. Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. *Trans. Info. For. Sec.* 12, 6 (June 2017), 1269–1284. https://doi.org/10.1109/TIFS.2017.2656460

[18] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.

[19] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *ICT Systems Security and Privacy Protection*, Hannes Federrath and Dieter Gollmann (Eds.). Springer International Publishing, Cham, 513–527.

[20] L. Li, T. F. Bissyande, and J. Klein. 2019. Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2901679

[21] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 318–329.

[22] Li Li, Tegawendé F. Bissyandé, Haoyu Wang, and Jacques Klein. 2018. CiD: Automating the Detection of API-Related Compatibility Issues in Android Apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 153–163. https://doi.org/10.1145/3213846.3213857

[23] L. Li, T. F. Bissyandé, J. Klein, and Y. L. Traon. 2016. An Investigation into the Use of Common Libraries in Android Apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 403–414.

[24] Li Li, Jun Gao, Tegawendé F. Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2018. Characterising Deprecated Android APIs. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 254–264. https://doi.org/10.1145/3196398.3196419

[25] Li Li, Jun Gao, Tegawendé F. Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2020. CDA: Characterising Deprecated Android APIs. *Empirical Software Engineering* 25, 3 (2020), 2058–2098.

[26] Li Li, Timothée Riom, Tegawendé F. Bissyandé, Haoyu Wang, Jacques Klein, and Le Traon Yves. 2019. Revisiting the impact of common libraries for android-related investigations. *Journal of Systems and Software* 154 (2019), 157–175. https://doi.org/10.1016/j.jss.2019.04.065

[27] Yu-Cheng Lin. 2015. AndroBugs Framework: An Android Application Security Vulnerability Scanner. In *Blackhat Europe 2015*.

[28] Damien Octeau, Somesh Jha, Matthew Dering, Patrick Mcdaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. 2016. Combining Static Analysis with Probabilistic Models to Enable Market-Scale Android Inter-component Analysis. In *Proceedings of the 43th Symposium on Principles of Programming Languages (POPL 2016)*.

[29] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*.

[30] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium*.

[31] Jordan Samhi, Alexandre Bartel, Tegawendé F. Bissyandé, and Jacques Klein. 2021. RAICC: Revealing Atypical Inter-Component Communication in Android Apps. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*.

[32] Xiaoyu Sun, Li Li, Tegawendé F. Bissyandé, Jacques Klein, Damien Octeau, and John Grundy. Accepted for publication on Nov. 29, 2020. Taming Reflection: An Essential Step Towards Whole-Program Analysis of Android Apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (Accepted for publication on Nov. 29, 2020), 1–20.

[33] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers* (Toronto, Ontario, Canada) *(CASCON '10)*. IBM Corp., USA, 214–224. https://doi.org/10.1145/1925805.1925818

[34] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2014. 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*. 1329–1341.

[35] Yanjie Zhao, Li Li, Haoyu Wang, Haipeng Cai, Tegawendé F. Bissyandé, Jacques Klein, and John Grundy. Accepted for publication on Jan. 07, 2021. On the Impact of Sample Duplication in Machine Learning based Android Malware Detection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (Accepted for publication on Jan. 07, 2021), 1–20.