AndroCT: Ten Years of App Call Traces in Android

Wen Li
Washington State University, USA
li.wen@wsu.edu

Xiaoqin Fu
Washington State University, USA
xiaoqin.fu@wsu.edu

Haipeng Cai
Washington State University, USA
haipeng.cai@wsu.edu

***Abstract*—Data-driven approaches have proven to be promising in mobile software analysis, yet these approaches rely on sizable and quality datasets. For Android app analysis in particular, there have been several well-known datasets that are widely used by the community. However, there is still a lack of such datasets that represent the run-time behaviors of apps—existing datasets are largely static, whereas run-time datasets are essential for data-driven dynamic and hybrid analysis of apps. In this paper, we present *AndroCT*, a large-scale dataset on the run-time traces of function calls in 35,974 benign and malicious Android apps from ten historical years (2010 through 2019). These call traces were produced by running each sample app against automatically generated test inputs for ten minutes. Moreover, each app was exercised both on an emulator and a real device, and the traces were separately curated. *AndroCT* has been used to build a novel dynamic profile of Android apps that has enabled several effective techniques and informative empirical studies concerning Android app security. We describe what this dataset includes, how it was created and stored, and how it has been used in past and would be used in the future.**

***Index Terms*—Android apps, dataset, function calls, tracing**

I. Introduction

The open nature of Android embraces framework-based app development and supports rapid reuse-based creation of user apps that provide ever-enriching functionalities. Meanwhile, the openness of Android contributes to the rise of varied challenges to developing, maintaining, and securing Android apps. For instance, accompanying the popularity of Android have been tremendous security threats and attacks targeting this dominant platform, resulting in the proliferation of security threats in the Android ecosystem [1]. Also, their peculiar behavioral traits and programming practices lead to unique difficulties in testing and debugging Android apps [2]–[4].

In response, researchers have developed techniques [5] for and conducted studies [5]–[7] on Android apps to better assure their quality. Among others, data-driven approaches have gained great momentum, including learning-based app classification [8]–[10] and bug fixing based on extracting behavioral patterns from known examples/samples [3], [11]. Essential to these technical approaches is the availability of a sizable and informative dataset. Such datasets can also serve as an immediate basis to empirical approaches that mine new insights about apps. Several datasets (e.g., Malgenome [12], Drebin [13], AMD [14], RmvDroid [15], and AndroZooOpen [16]) have been curated and made publicly available. Especially, the AndroZoo project [17] offers a continuous service collecting millions of apps from a variety of sources. These efforts have enabled a growing number of techniques and studies that advanced our knowledge about the Android ecosystem.

Meanwhile, existing *publicly available* datasets on Android apps have one or more of the following shortcomings.

- First, the majority of them are exclusively focused on malware, as mainly motivated by the need for developing security defense solutions (e.g., malware detection and categorization). Thus, these datasets are not sufficient for techniques/studies that require data on benign apps.
- Second, most of them just provide the apps themselves, with a few others including metadata (e.g., AndroZoo, RmvDroid), but they do not provide derived data (e.g., code traits of apps) that would require substantial computation costs to generate. Thus, techniques/studies that need the derived data would incur the costs repeatedly.
- Third, existing datasets are largely static (e.g., describing the app code), including no information about the run-time behaviors of apps. These datasets are thus not sufficient for dynamic techniques/studies.
- Fourth, most current datasets span a relatively short range of time—for instance, 4 years with RmvDroid, 3 years with Malgenome, and 1 year with Drebin. Thus, these datasets may not be sufficient for a longitudinal, evolutionary study (which would need to examine datasets spanning a much longer period of time). A few datasets, such as AMD and AndroZoo, do span more years but still they are static in nature.

To complement existing datasets on Android apps and to overcome their shortcomings as stated above, this paper presents *AndroCT*, a large-scale longitudinal dataset of 35,974 Android apps from ten years (2010 through 2019). In addition to reflecting an evolutionary perspective in sampling the Android app population, our dataset includes not only the benchmarks themselves but more importantly a major type of data on run-time app behavioral characteristics—the trace of calls to any functions in an app, including functions in exception-handling constructs and those invoked through reflection. Each app trace was generated by running the app against randomly generated run-time inputs for ten minutes, and was ensured to cover the majority of the app's code.

It is well known that Android apps, especially malicious ones, can exhibit different behaviors across different run-time environments (e.g., emulators versus real devices)—for instance, anti-dynamic-analysis malware can hide their malicious/suspicious behaviors when they detect that they are

running on an emulator. Considering these differences and the research value of enabling deep understanding of such differences, we further collected call traces of each app on both an emulator and a real Android device with the same Android platform version, for the same length of time and same run-time input coverage criteria. The entire *AndroCT* dataset took over 12,000 net machine-hours to produce on high-performance servers with large memory. *AndroCT* also includes two reasonably balanced subsets: one for 18,277 benign apps and the other for 17,697 malicious apps.

While typically only the original/raw datasets are necessary and derived data should be computed on demand, we aim to share our app execution traces as a dedicated kind of dataset in order to save the time that would be incurred by a purely dynamic or hybrid technique/study that needs these derived data. By doing so, our vision is that this dataset would boost the research on dynamic or hybrid technical and empirical app analysis. Not only is our dataset useful for developing data-driven approaches to app testing, debugging, and security analysis, it also opens doors for knowledge discovery about app run-time behaviors through empirical studies.

A peculiar merit of *AndroCT* lies in its inclusion of both emulator-based and real-device-based collections of dynamic call traces of a large number of apps—not only can each collection serve respective needs (e.g., emulator-based dynamic malware detection versus on-device real-time malware detection), the differences between these two collections are highly valuable for anti-dynamic-analysis malware defense research. For instance, a learning-based approach can developed using *AndroCT* to detect whether an app has evasive behaviors. Although function calls do not give the full spectrum of an app's run-time profile, they do provide a useful way to understand and model app behaviors.

Another value of our dataset comes from its longitudinal construct. *AndroCT* contains the call traces of apps from ten different years, a span of a non-trivial length to suffice for studying any of the above-mentioned app behavioral traits and security properties in an evolutionary perspective. In particular, both the differences between malware and benign apps, and those between emulator-driven and real-device-driven calling relationships in apps, may form the basis of various insights into sensitive, malicious, and evasive behaviors of apps.

The originality of *AndroCT* is three-fold. First, the entire dataset has never been used before. Relevant prior works only used part of the datasets, corresponding to the apps of years 2010 through 2017, and used that part in one way—through a particular behavioral profile defined by a specific set of metrics of the function call traces. Second, the entire dataset has never been comprehensively described and documented as in *AndroCT*, especially in terms of the inclusion of both emulator- and real-device-based traces. Third, we are not aware of any prior open dataset focusing on dynamic, derived data of apps based on function call traces, despite the presence of a number of benchmark suites and static datasets. To the best of our knowledge, *AndroCT* represents the first longitudinal Android app dataset on the dynamic call traces of both benign and malicious apps for as long as ten years. The *AndroCT* dataset has already been made publicly accessible at

https://zenodo.org/record/4470320#.YBEtnehKhPY

TABLE I
DATA SOURCES AND STATISTICS

	Benign apps		**Malware**	
Year	*#apps*	*source*	*#apps*	*source*
2010	1,530	AndroZoo	2,029	AndroZoo
2011	2,019	AndroZoo	2,431	AndroZoo
2012	2,053	AndroZoo	2,225	AndroZoo
2013	1,748	AndroZoo	1,230	Virusshare
2014	3,127	AndroZoo	1,493	Virusshare
2015	1,333	AndroZoo	1,667	Virusshare
2016	1,548	AndroZoo	2,171	Virusshare
2017	2,093	GooglePlay	1,935	AndroZoo
2018	1,465	GooglePlay	1,410	AndroZoo
2019	1,361	GooglePlay	1,106	AndroZoo
2010–2019 (total)	18,277		17,697	

II. THE *AndroCT* DATASET

The *AndroCT* dataset includes derived data for at least one thousand apps, malicious or benign, from each of ten past years. The statistics on the apps and sources from where the apps were originally obtained are outlined in Table I. *AndroCT* does not include these apps (e.g., their APKs) themselves for two reasons. The first is the large space cost of these apps combined and the storage limit with the free, public online data repositories. The second is that in our dataset the APK name of each app is given, according to which all of the apps can be readily downloaded from respective sources. As shown, the benign apps of years 2017 through 2019 were downloaded from the Google Play Store [18] and the malware of years 2013 through 2016 was obtained from VirusShare [19]; other apps were all from AndroZoo [17]. The noticeable variations among the sizes of yearly datasets (with the minimum of 1,106 and maximum of 3,127) were a result of applying our app selection criterion as detailed in Section III.

The derived data in *AndroCT* consists of two collections of function call traces. Each collection is composed of 35,974 trace files each corresponding to the trace of function calls in a 10-min continuous exploration of one of the 35,974 apps summarized in Table I. The only difference between these two collections is that one was produced on an Android emulator while the other on a real Android device (smartphone).

For each app, the execution trace in either collection, as stored in a text file, contains a number of textual line each representing a function call in the format of "f->g" where f is the caller and g is the callee. Both f and g are full-qualified method signatures, prefixed by the full class path of respective methods while including each parameter and return data type if any. An real example line of call trace is "`<info.universalmetadata.android.apps.novelreader.VoidLayout: void onCreate(android.os.Bundle)> -> <android.view.ContextThemeWrapper: android.content.res.Resources getResources()>`".

Note that the function call traces that we curated in *AndroCT* were all whole-app traces, meaning that we profiled invocations of all methods in the app—for methods whose

definition cannot be found from the app's APK, they were profiled only as callees. In terms of the scope of tracing, these methods cover all of the three code layers of an Android app—*user code*, *Android SDK*, and *third-party libraries*, and they may be defined in an app component of any of the four standard types: *Activity*, *Service*, *BroadcastReceiver*, and *ContentProvider*. In terms of the invocation mechanism of the calls, these methods include both those called explicitly (i.e., with explicit references to the call targets) and those that are the targets of reflective calls, and they may be called through normal or exceptional control flows.

Especially, when a callee is an inter-component communication (ICC) API, the trace contains the value of each ICC Intent field, immediately after the line about the call itself. For non-ICC calls, currently *AndroCT* does not trace the return values or the values of arguments at a callsite. The reason that we chose to profile more details only for ICC calls was that without these Intent details it would be hard to link the communicating components. Establishing these links would be essential for whole-app dynamic analysis [20].

Since we have apps in both malware and benign-app classes and across ten years, while in two separate collections, *AndroCT* includes 2*10*2=40 subdatasets. In our data package, each of these 40 subdatasets is stored as one zip file (.tar.gz), with the file name explicitly indicating of which year and in which security category (`malware/benign`) the underlying apps are. Moreover, for the collection generated on the real device, the name is further prefixed by the word "real".

III. Data Collection and Construction

We describe how we collected the apps (as summarized in Table I) and how the *AndroCT* dataset was constructed (i.e., how the function call trace was curated for each app).

A. App Collection

As it alone provides both benign and malicous apps of different years from various origins, we took AndroZoo as the main source of apps. To further diversify the data sources for better sample representativeness, we considered Virusshare and Google Play as two alternative data sources as well.

For each year, we started with downloading a pool of apps and discarded those for which we cannot meaningfully profile their function calls (due to corrupted APK or failed instrumentation) or cannot attain 60% or higher line coverage of the app's user code after running the app against test inputs randomly generated by Monkey [21] for ten minutes. We used these selection criteria because our focus is on the run-time behavioral profiling of apps in terms of function calls, and 60% coverage enforces reasonable confidence about the profile's representativeness of app behaviors. The Monkey tool was used because of its industry-strength robustness and usability as well as relative small shortage in code coverage compared to various research prototypes for Android app testing. For all the 35,974 apps selected eventually, the mean line coverage per app is 74.85% (with standard deviation 11.97%). We did not intent to curate balanced numbers of apps across years given the uneven distribution of the total app populations over the years. We also removed redundant apps within and across the ten years. Further, a benign sample was removed if at least one VirusTotal [22] tool identified it as malware, and a malware sample was removed if less than ten of VirusTotal tools agreed on its status of being malicious.

B. Function Call Tracing

The whole-app function calls for each apps were all traced using *DroidFax* [23]. To generate the trace, the tool performed static instrumentation purely at app level to probe for the calls of any method, including ICC APIs, reflective calls, and calls invoked through exceptional control flows (e.g., in a `catch` or `finally` block). For ICC API calls, additional probes for profiling the underlying Intent object were inserted.

For the emulator-based trace collection, each instrumented app was then explored by Monkey for ten minutes on a Google Nexus One emulator of 2G RAM and 1G SD storage with the Android 6.0 installed. The execution trace was collected at the host machine through the Logcat tool [24]. To avoid possible interferences among the executions of different apps, the emulator was restarted as a fresh clean environment before tracing the next app. Further details on the instrumentation and run-time profiling can be found in [23].

For the real-device-based trace collection, we followed the same process as for creating the emulator-based trace collection, reusing the same instrumented apps. The only difference was that here we ran each app on a Samsung Galaxy S4 smartphone with the same Android platform version, 2G RAM, and 4G SD storage.

IV. Prior Use of *AndroCT*

For conducting a dynamic characterization study [25], we proposed an app behavioral profile, defined by 122 metrics in three dimensions as described in [20]—these metrics were immediately computed from the function call traces in *AndroCT* although only 125 benign apps of a single year were used in the study. In defining these metrics, we differentiated function calls in different scopes (e.g., the three code layers and four types of app components as mentioned earlier). We also categorized callbacks, a subset of function calls, into various categories. Moreover, we separately treated calls to functions that are pre-defined sources and sinks, while using various source/sink categories to define relevant metrics. The study was later extended to include apps of years 2010 through 2017 [26], but still only benign apps were considered.

In addition, leveraging another subset of *AndroCT*, we have built the same dynamic profiles but only for 136 benign apps and 135 malware, both of a single year. From these profiles, we discovered 70 out of the 122 metrics that best differentiated the dynamic profiles between malware and benign apps, and utilized these 70 metrics to develop a new dynamic Android malware detection and categorization tool [27]. Furthermore, we have utilized the *AndroCT* dataset for developing and evaluating sustainable malware detectors [10], [28], [29]—the longitudinal nature enabled us to assess the sustainability of

the detector in classifying apps appeared several years after the year of the apps used for training.

Most recently, we utilized part of *AndroCT* to perform a comparative study of the behavioral differences between malware and benign apps, in terms of the same behavioral profile. The study also only used apps of years 2010 through 2017 [30]. To illustrate the use of *AndroCT* in constructing this behavioral profile, the current data package of *AndroCT* includes such profiles of apps as used in that piror paper.

V. Future Use of *AndroCT*

Despite the multiple prior studies that have used *AndroCT*, the entire dataset has not been utilized as a whole. In particular, the real-device-based trace collection is largely untapped—only the traces of benign apps of years 2010 through 2017 were used in [26]; all of the other studies only used the emulator-based traces. More importantly, all the prior studies represented only one application of *AndroCT* in terms of dataset construction—they all just used the traces to build the single particular behavioral profile defined by the 122 metrics mentioned before, and none of them have used the traces directly or in other ways (e.g., constructing a different run-time profile of Android apps). Also, the traces of apps of years 2018 and 2019 have not yet been used in any fashion. These instances of underuse open a range of opportunities to leverage *AndroCT* for many more potentially significant research works in the future, as exemplified below.

First, the entire dataset of *AndroCT* can be used to extend the prior studies by examining the longer-span (eight- versus ten-year) evolution of app behaviors, while looking at the differences in app behaviors between virtual (emulator) and real execution environments. Thus, in terms of the same behavioral profile, we may answer questions like *how do Android apps behave differently on an emulator versus on a real Android device?*, *how do benign apps behave differently from malware on a real device?*, and *how do these differences differ from those obtained on an emulator?*. Each of these questions can be further studied under an evolutionary lens. We can also answer whether the evolutionary patterns sustain beyond a certain span, by comparing the patterns observed in our prior works with the patterns in the entire dataset.

Second, *AndroCT* can be used in ways other than enabling the computation of the 122 metrics used before. Many other different metrics and measures that represent/model app behaviors in terms of function calls can be proposed. For example, dynamic call graphs can be constructed from the call traces and traversing the graphs will lead to (function-level) execution paths; then known graph and sequence metrics/measures can be computed. For another example, particular kinds of app behaviors can be studied from the call traces by looking at special kinds of calls (e.g., characterizing reflective calls and exceptional handling calls). Various data mining techniques may also be applied immediately to the traces to mine general and special call patterns in Android apps. All in all, based on these novel behavioral profiles, questions similar to the above-mentioned ones can be answered to generate additional, novel insights into the run-time behaviors of Android apps.

Third, the differences between the two collections in *AndroCT* immediately enable studies of the potentially evasive nature of app behaviors in terms of function calls and ICCs. Since these two collections differ only in the underlying execution environments, the differences between the same apps intuitively reflect the existence and patterns of evasive apps that only exhibited malicious or suspicious behaviors on real devices but hid those behaviors in a virtual execution environment like emulators. Thus, *AndroCT* would help answer security defense related questions like *what are the typical functional call and ICC patterns of evasive apps?*, *how have the evasion schemes adopted by such apps evolved in terms of function calls and ICCs?*, and *is malware significantly more evasive than benign apps, or are benign apps not evasive at all?*. Using *AndroCT*, we can also develop learning-based detectors of anti-dynamic-analysis schemes adopted in Android apps and apps that adopt such schemes. Moreover, *AndroCT* can be immediately used in evaluating different kinds of such detectors. In particular, the longitudinal nature of *AndroCT* makes it especially suitable for assessing the sustainability [10] of learning-based detectors of evasive malware.

None of these above questions have been studied by ourselves, nor have they been addressed by others as we are aware of. Thus, we believe *AndroCT* has great potential and impact for future relevant research.

VI. Challenges, Limitations, and Improvement

We faced two major challenges when creating *AndroCT*. The first was the high overhead of profiling a sizable set of apps—tracing each app took 10 minutes. The second was the difficulty of reaching the size of nearly 36,000 while applying the selection criteria—a number of apps were discarded (e.g., as they cannot be instrumented, the instrumented versions did not run, or the 10-min executions did not cover 60% or more of the app's code) before an app was selected.

As a result of these challenges, one limitation of *AndroCT* is that the call traces only represent a limited code coverage for each app hence may not reflect the typical run-time behaviors of the app. Another limitation is that the current dataset only represents one particular emulator and one real device with one Android platform version. When applied to empirical studies, especially those of an evolutionary view, an additional limitation concerns the representativeness of the apps of each year as opposed to the entire app population of that year.

There are several ways to expand and improve *AndroCT*. First, the function call traces can be much enriched by including run-time values of variables harvested during app executions, including the return values and function arguments. These values will enable more fine-grained modeling of app behaviors. Second, the representativeness of app behaviors by *AndroCT* can be improved by using higher-quality test inputs (e.g., as generated by state-of-the-art fuzzing tools applicable to Android apps). Finally, the dataset can be enlarged in size for each year and expanded to cover more years.

REFERENCES

[1] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: a survey of issues, malware penetration, and defenses," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[2] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for Android: Are we there yet?" in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 429–440.

[3] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 187–198.

[4] H. Cai, "Embracing mobile app evolution via continuous ecosystem mining and characterization," in *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, 2020, pp. 31–35.

[5] K. Holl and F. Elberzhager, "Mobile application quality assurance," in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 1–77.

[6] H. Cai, Z. Zhang, L. Li, and X. Fu, "A large-scale study of application incompatibilities in Android," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[7] Z. Zhang and H. Cai, "A look into developer intentions for app compatibility in Android," in *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft)*, 2019.

[8] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of Android malware and Android analysis techniques," *ACM Computing Surveys*, vol. 49, no. 4, p. 76, 2017.

[9] H. Cai, "A preliminary study on the sustainability of android malware detection," *arXiv preprint arXiv:1807.08221*, 2018.

[10] ——, "Assessing and improving malware detection sustainability through app evolution studies," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2019.

[11] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, "Vurle: Automatic vulnerability detection and repair by learning from examples," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 229–246.

[12] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. of the IEEE Symposium on Security and Privacy*, 2012, pp. 95–109.

[13] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Efficient and explainable detection of Android malware in your pocket," in *Proceedings of Network and Distributed System Security Symposium*, 2014.

[14] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 252–276.

[15] H. Wang, J. Si, H. Li, and Y. Guo, "Rmvdroid: towards a reliable android malware dataset with app metadata," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 404–408.

[16] P. Liu, L. Li, Y. Zhao, X. Sun, and J. Grundy, "Androzooopen: Collecting large-scale open source android apps for the research community," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 548–552.

[17] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of Working Conference on Mining Software Repositories*, 2016, pp. 468–471.

[18] Google, "Google Play Store," 2020, https://play.google.com/store?hl=en.

[19] virusshare.com, "VirusShare," 2020, https://virusshare.com/.

[20] H. Cai and B. G. Ryder, "Artifacts for dynamic analysis of android apps," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 659–659.

[21] Google, "Android Monkey," http://developer.android.com/tools/help/monkey.html, 2017.

[22] virustotal.com, "VirusTotal," 2017, https://www.virustotal.com/.

[23] H. Cai and B. Ryder, "DroidFax: A toolkit for systematic characterization of Android applications," in *Proceedings of International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 643–647.

[24] Google, "Android logcat," http://developer.android.com/tools/help/logcat.html, 2015.

[25] H. Cai and B. Ryder, "Understanding Android application programming and security: A dynamic study," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 364–375.

[26] ——, "A longitudinal study of application structure and behaviors in Android," *IEEE Transactions on Software Engineering (TSE)*, 2020.

[27] H. Cai, N. Meng, B. Ryder, and D. D. Yao, "Droidcat: Effective Android malware detection and categorization via app-level profiling," *IEEE Transactions on Information Forensics and Security (TIFS)*, 2019.

[28] H. Cai and J. Jenkins, "Towards sustainable android malware detection," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 350–351.

[29] X. Fu and H. Cai, "On the deterioration of learning-based malware detectors for Android," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2019, pp. 272–273.

[30] H. Cai, X. Fu, and A. Hamou-Lhadj, "A study of run-time behavioral evolution of benign versus malicious apps in Android," *Information and Software Technology (IST)*, 2020.