TwinDroid: A Dataset of Android app System call traces and Trace Generation Pipeline

Asma Razagallah
Univeristé du Québec à Chicoutimi
Saguenay, Québec, Canada
asma.razgallah1@uqac.ca

Raphaël Khoury
Univeristé du Québec à Chicoutimi
Saguenay, Québec, Canada
raphael.khoury@uqac.ca

Jean-Baptiste Poulet
Univeristé du Québec à Chicoutimi
Saguenay, Québec, Canada
jean-baptiste.poulet1@uqac.ca

ABSTRACT

System call traces are an invaluable source of information about a program's runtime behavior and be particularly useful for malware detection in Android apps. However, the paucity of publicly available high-quality datasets hinders the development of the field. In this paper, we introduce *TwinDroid*, a dataset of over 1000 system calls traces, from both benign and infected Android apps. A large part of the apps used to create the dataset is from benign-malicious app pairs, identical apart from the inclusion of malware in the latter. This makes *TwinDroid* an ideal basis for security research, and an earlier version of *TwinDroid* has already been used for this purpose. In addition to a dataset of traces, *TwinDroid* includes a fully automated traces generation pipeline, which allows users to generate new traces in a standardized manner seamlessly. This pipeline will enable the dataset to remain up-to-date and relevant despite the rapid pace of change that characterizes Android security.

KEYWORDS

dataset, Android apps, security, System call traces

ACM Reference Format:
Asma Razagallah, Raphaël Khoury, and Jean-Baptiste Poulet. 2022. TwinDroid: A Dataset of Android app System call traces and Trace Generation Pipeline. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524842.3528502

1 INTRODUCTION

System call traces reveal a wealth of information about the execution of the underlying system and serve as the basis for a variety of analyses, including malware detection, optimization, feature enhancement, and performance analysis. System call traces have been shown to be particularly useful in the case of malware detection in Android apps, and a number of security solutions that rely upon system calls have been proposed [10], [1], [9], [15], [17].

However, the paucity of publicly available datasets of Android traces— often decried by researchers, forces each research team to rely upon its custom-generated dataset, with the consequence that it is not easy to compare the effectiveness of these methods on equal footing. The considerable time required to generate an adequate volume of traces is also an obstacle to developing security solutions. Many studies on the topic are tested on trace sets whose size seems insufficient to draw general conclusions.

To address these issues, we present *TwinDroid*, a dataset of Android system call traces. At present, *TwinDroid* contains 10,859 traces. A large part (42%) of the traces present in the dataset has been generated from pairs of benign and infected (repackaged) versions of the apps. The presence of apps that are identical but for the inclusion of malware makes this dataset ideal for research on dynamic malware detection. We intend to continue to expand the size and variety of the dataset in the near future.

Furthermore, *TwinDroid* repository also includes a well-controlled and reproducible pipeline, which automates the process of generating new traces. This allows users to seamlessly generate new traces that are tailored to their particular needs. For example, a researcher who wishes to study the evolution of Android apps over a period of time may generate traces from both earlier and more recent apps.

In addition to traces, *TwinDroid* also extracts permissions and features from each app. These datums are highly predictive of malicious behavior and repackaging [12], [19],[5].

While specially designed for security purposes, *TwinDroid* is sufficiently versatile to serve in research in several areas of software engineering, mainly because of the ability to generate additional traces on-demand with minimal effort and allows the dataset to remain up to date.

TwinDroid offers several advantages, namely:

- The dataset consists of traces of apps drawn from multiple sources and contains both benign apps and infected apps with various malware. An important part of the dataset consists of traces from pairs of applications, one benign, one malicious, but identical but for inclusion of the malware in the latter. This makes TwinDroid uniquely suited for security research.
- The apps traced span a variety of common app categories and have been published over ten years. For each app, the dataset contains multiple traces of varying lengths.
- In addition to each trace, the database records the random seed and event stream that created the trace. This provides a large degree of reproducibility and a path for research on explainability if a trace exhibits interesting or unusual behavior.
- In addition to the dataset of existing trace, *TwinDroid* presents a fully automated pipeline to generate new traces from existing apps. Furthermore, any researcher in need of app traces may use this pipeline to generate traces suitable to their specific needs. This ease of extension is a crucial design goal

MSR '22, May 23–24, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3528502

as it allows the database to remain up-to-date with emerging malware.

An earlier version of the *TwinDroid* dataset was used to perform malware detection on abstracted traces [13].

Both the dataset and the trace generation pipeline are public and freely available for research purposes[1] [2].

The remainder of this paper is organized as follows: Section 2 describes the dataset, while Section 3 details the associated trace generation pipeline. Section 4 discussed some limitations of the dataset, and Section 5 presents some avenues of research that TwinDroid opens.

2 THE TWINDROID DATASET

The *TwinDroid* dataset includes over 10,859 traces, from 773 benign and 4,715 infected apps. Each trace is generated by using the Monkey tool[3] to run the app on random inputs for a pre-determined amount of time (the dataset includes traces of differing lengths) and tracing the execution using Strace[4]. All apps are run in an emulated environment.

TwinDroid's database schema is given in Figure 1.

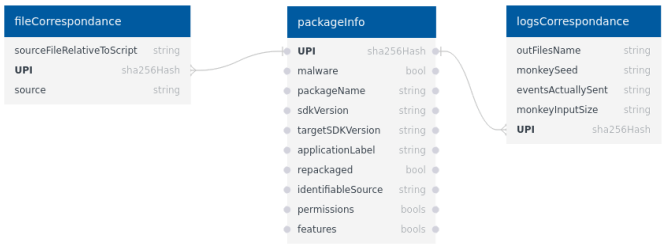

Figure 1: Database Schema

A Unique Package Identifier (UPI), computed by applying the SHA-256 hash to each app, serves as a primary key (field `Unique Package Identifier`). The fileCorrespondance table additionally contains the relative path of the apk on the host system, including its filename (field `sourceFileRelativeToScript`) and the source of the app, in case apps are selected from multiple datasets of apps.

Table packageInfo contains a variety of information about the apps. Notably, this table indicates the file name of the apk (field `packageName`), the version of the SDK (field `sdkVersion`) and the target SDK version (field `TargetSDKVersion`). this information is extracted from the manifest file, as is the name of the app (field `ApplicationLabel`). The presence or absence of malware is indicated by the Boolean attribute `malware`. In many cases, the exact name of the malware infecting an apk can be obtained from the database from which the apk was downloaded.

The `permissions` and `features` fields respectively contain the permissions and features declared in the manifest file of the apk. This information was included in the database because of its relevance to malware detection and other security-relevant applications. These fields are constructed by comparing the permissions and features declared in the manifest file with lists contained in two files, "features.txt" and "permissions.txt", which include all standard permissions and features listed on the official Android developer documentation at the time of the creation of TwinDroid. This scheme was adopted to avoid cluttering the database with user-defined permissions, which are not informative. When generating additional traces, the user may update these files to reflect any changes in the standard permission and feature sets.

Finally, the logsCorrespondance table holds data resulting from the actual tracing. For a given UPI, multiple entries will exist, one for each trace test performed on the apk. The field `OutFilesName` holds the base name of the two files generated for each trace: a `.monkdata` file that contains the output of the Monkey script and a `.trace` file that contains the trace proper. To improve reproducibility, we also include the seed passed as a parameter to the Monkey tool (field `MonkeySeed`), to generate the random input as well as the number of input that was intended to be sent by Monkey to the app (field `MonkeySizeInput`). The field `eventActuallySent` contains the number of events that the Monkey tool was actually able to send before the application failed or execution was terminated. Such failures are usually caused by a crash in the application. This is not surprising considering that the script performs fuzzing, which much ill-developed software can't handle. Such failures do not hinder the execution of the tracing process: the abnormal execution is recorded in the database, and the tracing proceeds. If `monkeySizeInput` and `eventActuallySent` are equal it can be assumed that the test was successfully completed.

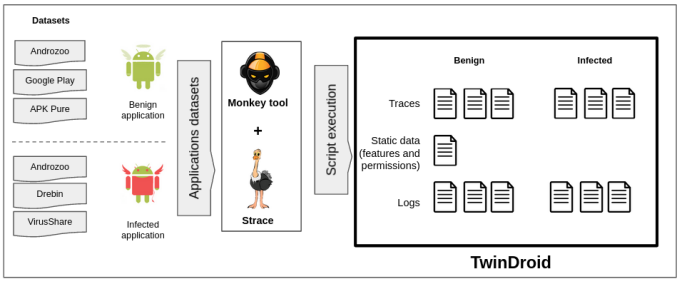

Figure 2: Overview of the Trace Generation Pipeline.

Each trace consists of two files, a trace file, and a .monkeydata file.

The trace file is the output of Strace, with parameters: -f -ttt. These parameters can be modified if additional traces are generated.

Each line of the trace file corresponds to a single system call and indicates a timestamp, followed by the system call itself, including parameters and return values. If more than one thread or process is traced (because the initial process forked), then the PID of the process that performs the system call is prefixed to each line. A sample of a trace is shown in Figure 3.

The .monkdata file is the output of the Monkey tool on stdout. It contains a wealth of data that aids in understanding the underlying behavior of the trace, notably a listing of the random events sent to the app. We refer the reader to the Monkey tool's documentation for more information on this data.

[1] https://github.com/RaphaelKhoury/automated-apk-tracing
[2] https://zenodo.org/record/6259612#.YioFmxvCpH4
[3] https://developer.android.com/studio/test/monkey
[4] https://strace.io/

Table 1: Content of the *TwinDroid* dataset.

Dataset	Benign apps	Malicious apps	Pairs	Benign traces	Malicious traces	Years
Androzoo	704	1601	1601	1313	3201	2016 – 2021
Google Play	40	0	22	80	0	2008 – 2022
APKpure	29	0	0	38	0	2014 – 2022
Drebin	0	3114	22	0	6227	2010 – 2012
Total	773	4715	1623	1431	9428	

```
[pid  3103] 1641327580.279530 prctl(PR_SET_VMA, PR_SET_VMA_
ANON_NAME, 0x713d39b38000, 16384, "stack_and_tls:3103") = 0
[pid  3103] 1641327580.279611 mmap(NULL, 36864, PROT_READ|
PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x713d39aad000
[pid  3103] 1641327580.279688 mprotect(0x713d39aad000, 4096,
PROT_NONE) = 0
[pid 3103] 1641327580.279750 sigaltstack({ss_sp=0x713d39aae000,
ss_flags=0, ss_size=32768}, NULL) = 0
[pid 3103] 1641327580.279820 prctl(PR_SET_VMA, PR_SET_VMA_ANON_
NAME, 0x713d39aae000, 32768, "thread signal stack") = 0
[pid 3103] 1641327580.280081 rt_sigprocmask(SIG_SETMASK, [QUIT
USR1 PIPE RTMIN], NULL, 8) = 0
[pid  3103] 1641327580.280175 mprotect(0x713a96c04000, 4096,
PROT_NONE) = 0
```

Figure 3: Sample of an application trace

Each trace is on average *4kB*, for a total of *7GB* for the entire dataset.

Origin of the apps. The TwinDroid dataset currently contains over 10,859 traces from 4,715 malicious and 773 benign apps. Of these, 2,327 apps occur in matching pairs of otherwise identical benign and malicious apps. The make-up of the dataset is detailed in Table 1.

We opted to include samples of traces from several different sources, both to increase the diversity of app types and malware types present in the dataset, as well as to illustrate the ease with which new traces can be added using the trace generation pipeline. As such, we selected benign apps from the Google play [8], and APKpure [3] datasets, and malicious apps from the Drebin, [4] Andozoo [2] and Google play datasets. The Androzoo dataset identifies 15,000 repackaged variants of 3,000 benign apps in the Androzoo dataset. An additional 22 pairs were manually identified by comparing the Google Play and Drebin datasets. We used the online tool VirusTotal[5] to determine if the applications of Google Play and Pure APK are benign.

All of these datasets are widely used in academic research. Also, note that while Androzoo and APKpure contain more recent apps appropriate for security research, those from Google play and Drebin span a more extended period, allowing longitudinal studies on software engineering.

[5]https://www.virustotal.com/gui/home/upload

3 TRACE GENERATION PIPELINE

In addition to the traces already present, TwinDroid includes a trace generation pipeline that automates the process of tracing Android apps and of generating new trace data. The apps are run on an emulator, which is run inside a container. This design choice is motivated by providing identical settings for all trances. Using a container avoids the time-consuming process of installing and configuring the emulator since the container was created from a system on which the emulator is already installed. The use of containers provides additional benefits, including :

- Additional security and isolation of the malware from the underlying system;
- Ease of deployment, avoiding the installation of the emulator, and providing a common emulator setup for every execution;
- Possible parallelization.

The pipeline creates a fresh container for each app and then proceeds to install the apk via adb. Duplicate apks are seamlessly detected (by way of the hash) and discarded.

The pipeline first extracts static information from the apk (features and permissions) before tracing. Finally, each trace is added to the database upon completion.

If a trace of a given apk and input size already exists in the database, the script skips to the next input size or apk. This ensures that the script can be stopped at any point during tracing and then restarted, skipping over the work already done.

Implementation Considerations. To generate new traces, the user only has to update the configuration file and run script.sh. More specifically, the user must specify: (1) the number of traces to generate for each app, and the number of events that the Monkey tool should generate in each case; (2) the paths of the folders containing the benign and infected traces; (3) and optionally, a list of pairs of apps, identified by their hash values, in case the dataset includes matching pairs of benign and infected versions of the same app. The apps can be copied directly in the benign and infected folders. However, if the apps come from multiple sources and the user wishes to keep track of each app's origin, a distinct sub-folder for each app source should be created in these folders. The name of this sub-folder will populate the `source` field in the database.

As mentioned above, both the Monkey tool and the emulator are run inside an emulator. Unfortunately, Monkey launches the app internally and does not offer the possibility of simultaneously invoking the Strace. To circumvent this problem, it was decided to trace the Monkey process from the onset alongside all of its sub-processes, encapsulating the application to test. This solution is perhaps sub-optimal since it implies that one of the processes

being traced is the Monkey tool rather than the app. Still, the only plausible alternative, starting Strace after launching the app, would omit the beginning of the execution from the trace. Such a situation would be particularly undesirable since malware often performs malicious actions at the onset of the execution. In any case, it is straightforward to identify the system calls made by the Monkey tool, rather than by the underlying application, by their PID[6].

Some security risks are unavoidable when running apps infected with malware, even if this is done only in a simulated environment. In fact, emulators may themselves contain security vulnerabilities that could be exploited by the malware present in the underlying app [18].

The added layer of security provided by the container should thwart such attempts. The docker container runs without administrator privilege on the host machine, and it incorporates multiple built-in security features, which should ensure that the host system remains unaffected if the container is compromised. No security measure is perfect, but the attack surface should be limited to adb and the two exposed ports on the container.

4 LIMITATIONS

An important limitation of TwinDroid is the fact that the apps are run in an emulated environment. Some malware may be able to detect the presence of the emulator and will not perform malicious actions. The collected trace is thus not an accurate reflection of the actual behavior of the underlying malicious app, with a consequence for its suitability in malware detection.

In a small number of cases, the emulator is unable to run the app, with consequences for the completeness of the dataset if we require traces from every app in a sample. It is important to stress that this situation is comparatively rare, occurring in less than 5% of the apps we attempted to trace.

As discussed above, malicious apps are prone to crash in fuzzing, which reduces the utility of the traces. However, since our database contains, for each trace, the number of events that the Monkey tool intended to send, as well as the number of events that were sent, it is easy to exclude such misbehaving traces if they are unwanted.

5 PREVIOUS AND FUTURE USE

In our previous survey of malware detection in Android apps [14], we found that system calls were one of the preferred sources of information to profile malicious behavior in apps. However, the heterogeneity of the datasets commonly used in research makes comparing the effectiveness of different methods difficult. The considerable time required to generate a dataset of suitable size is also in hindrance to the development of the field. The use of *TwinDroid* can help address these issues.

In our subsequent research [13], we employed an early version of *TwinDroid* for malware detection in system call traces. An analysis of the pairs of benign and malicious traces present in *TwinDroid* allowed us to develop a trace abstraction process that highlights the differences between the two categories of traces. Performing malware detection on the abstracted traces was shown to improve detection accuracy.

[6]Note that Strace only records the PID after the first fork occurs. Previous system calls, for which a PID is not listed, are also performed by the Monkey tool.

As mentioned above, the fact that a large portion of TwinDroid consists of traces from pairs of otherwise identical benign and malicious apps gives us the means to study the low-level behavior of malware with a higher degree of precision than was previously possible. In particular, this dataset will make it possible to verify that malware detection algorithms really are detecting malware behaviors and are not simply sorting different apps based on other features.

Aside from security purposes, system traces are widely used for several other software engineering tasks, and TwinDroid is a suitable basis in this regard. In this regard, TwinDroid's trace generation pipeline will allow the dataset to remain up-to-date and relevant on a forward-going basis.

Amongst possible usages, we note the creation of pattern libraries that relate sequences of system calls to higher-order events (e.g., Opening a file or sending data to the network) [7]. Such methods allow users to extract more information from the traces and open the door to further avenues of research, thus increasing the value of the dataset.

Karan et al. [1] show that the power consumption of a device can be estimated from the system calls performed, while Tian et al. rely upon patterns of system calls to detect code plagiarism [17]. Karn et al. [11] showed that an analysis of the system calls performed by an IoT device can reveal the presence of crypto mining.

Syed et al. [16] use traces of system calls for remote attestation, the process of ensuring the trustworthiness of clients running software locally. Ezzati-Jivan [6] shows how system call traces can reveal the root cause of performance degradation in distributed systems.

Furthermore, the large time span of the apps traced in the dataset allows us to perform longitudinal studies on the evolution of these aspects of app development over the last several years.

REFERENCES

[1] Karan Aggarwal, Chenlei Zhang, Joshua Charles Campbell, Abram Hindle, and Eleni Stroulia. 2014. The power of system call traces: predicting the software energy consumption impact of changes.. In *CASCON*, Vol. 14. 219–233.

[2] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 468–471. https://doi.org/10.1109/MSR.2016.056

[3] APKPure. 2014-2022. https://apkpure.com/.

[4] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society.

[5] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '09)*. ACM, New York, NY, USA, 235–245. https://doi.org/10.1145/1653662.1653691

[6] Naser Ezzati-Jivan, Houssem Daoud, and Michel R Dagenais. 2021. Debugging of Performance Degradation in Distributed Requests Handling Using Multilevel Trace Analysis. *Wireless Communications and Mobile Computing* 2021 (2021).

[7] Waseem Fadel. 2010. Techniques for the Abstraction of System Call Traces to Facilitate the Understanding of the Behavioural Aspects of the Linux Kernel. https://users.encs.concordia.ca/~abdelw/papers/Fadel_MASc_S2011.pdf.

[8] Google. 2008. Google Play. https://play.google.com/store/apps?hl=fr.

[9] Lihong Han, Qingguo Zhou, Juheng Zhang, Xuhui Yang, Rui Zhou, and Jianxin Tang. 2020. Polymorphism and consistency: Complex network based on execution trace of system calls in Linux kernels. *International Journal of Modern Physics C* 31, 09 (2020), 2050126.

[10] Steven A Hofmeyr, Stephanie Forrest, and Anil Somayaji. 1998. Intrusion detection using sequences of system calls. *Journal of computer security* 6, 3 (1998), 151–180.

[11] Rupesh Raj Karn, Prabhakar Kudva, Hai Huang, Sahil Suneja, and Ibrahim M. Elfadel. 2021. Cryptomining Detection in Container Clouds Using System Calls and Explainable Machine Learning. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2021), 674–691. https://doi.org/10.1109/TPDS.2020.3029088

[12] Mengyu Qiao, Andrew H. Sung, and Qingzhong Liu. 2016. Merging Permission and API Features for Android Malware Detection. In *5th IIAI International Congress on Advanced Applied Informatics, IIAI-AAI 2016, Kumamoto, Japan, July 10-14, 2016*. 566–571. https://doi.org/10.1109/IIAI-AAI.2016.237

[13] Asma Razgallah and Raphaël Khoury. 2021. Behavioral classification of Android applications using system calls. *Proceedings of the 28th Asia-Pacific Software Engineering Conference, December 6-9, 2021* (2021), 73.

[14] Asma Razgallah, Raphaël Khoury, Sylvain Hallé, and Kobra Khanmohammadi. 2021. A survey of malware detection in Android apps: Recommendations and perspectives for future research. *Computer Science Review* 39 (2021), 100358.

[15] Jesper Simonsson, Long Zhang, Brice Morin, Benoit Baudry, and Martin Monperrus. 2021. Observability and chaos engineering on system calls for containerized applications in docker. *Future Generation Computer Systems* 122 (2021), 117–129.

[16] Toqeer Ali Syed, Salman Jan, Shahrulniza Musa, and Jawad Ali. 2016. Providing efficient, scalable and privacy preserved verification mechanism in remote attestation. In *2016 International Conference on Information and Communication Technology (ICICTM)*. IEEE, 236–245.

[17] Zhenzhou Tian, Ting Liu, Qinghua Zheng, Ming Fan, Eryue Zhuang, and Zijiang Yang. 2016. Exploiting thread-related system calls for plagiarism detection of multithreaded programs. *Journal of Systems and Software* 119 (2016), 136–148.

[18] Fenghao Xu, Siyu Shen, Wenrui Diao, Zhou Li, Yi Chen, Rui Li, and Kehuan Zhang. 2021. Android on PC: On the Security of End-user Android Emulators. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1566–1580.

[19] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*.