Software Vulnerabilities Detection Using a Trace-Based Analysis Model

Gouayon Koala[1(✉)], Didier Bassole[1], Telesphore Tiendrebeogo[2], and Oumarou Sie[1]

[1] Laboratoire de Mathématiques et d'Informatique, Université Joseph Ki -Zerbo, Ouagadougou, Burkina Faso
gouayonkoala1@gmail.com

[2] Laboratoire d'Algèbre, de Mathématiques Discrètes et d'Informatique, Université Nazi Boni, Bobo-Dioulasso, Burkina Faso
https://www.ujkz.bf, https://www.univ-bobo.gov.bf

Abstract. Over the years, digital technology has grown considerably. With this growth, the information systems' security has increasingly become a major concern. In this paper, we propose an analysis model based on application execution traces. This model makes it possible to improve the detection of vulnerabilities in applications. Indeed, after an evaluation of each of the tracing techniques we derived this model which takes into account these techniques and combines them with machine learning techniques. In this way, the applications undergo several analyses. This reduces the effect of evasion techniques used by hackers to circumvent the proposed solutions. We focused on Android applications because of their increasing popularity with a variety of services and features offered, making them a favourite target for hackers. These hackers use every means to exploit the slightest flaw in the applications. Unfortunately, the solutions proposed remain insufficient and sometimes ineffective in the face of their determination.

Keywords: Execution traces · Vulnerabilities · Tracing techniques · Applications · Attacks

1 Introduction

The role that applications play in computer systems makes them essential to progress. Applications use, although indispensable for companies and individuals, makes them a privileged target for malicious attacks [1]. These attacks exploit known or unknown vulnerabilities in the applications. The solutions proposed in the literature remain insufficient in view of the growth of attacks. These attacks and exploits are increasingly motivated by financial gain. These motives have led to changes in the nature of the attacks and also in the procedures used by cybercriminals. They are using techniques that make malware more sophisticated in order to evade the detection or protection systems in place [2].

R. A. Saeed et al. (Eds.): AFRICOMM 2022, LNICST 499, pp. 446–457, 2023.
https://doi.org/10.1007/978-3-031-34896-9_27

However, if applications contain flaws, these are exploited by cybercriminals to their advantage.

How do we combine the protection of application data with the continuous innovation of these applications? This is the question that mobilises researchers in their quest to solve these computer data security problems. They are committed to solving this problem. Thus, several avenues of solutions are being explored and several solutions are being experimented with in order to protect the data [2–5]. Unfortunately, their efforts come up against the determination of pirates. This means that the existing solutions are either insufficient or ineffective in the face of the rise of piracy techniques [1].

This study is part of the logic of finding solutions to reduce the effects of malicious actions on applications. For this purpose, we have chosen a dynamic approach based on the analysis of application traces during their execution. We propose a model for analysing these mobile application traces to improve data protection. The aim is to improve the detection of vulnerabilities in mobile applications, which will also improve data protection.

The rest of this document is divided into several sections. Section. 2 presents work similar to our study while Section. 3 provides a comparative study of different application tracing tools. In Section. 4 we describe our approach with a model based on the collection of application execution trace data. We conclude in Section. 5 with the conclusion where we present the synthesis of our contribution and our future work.

2 Related Works

2.1 Malware Evolution

According to the Gartner report [6], the number of applications has increased considerably in recent years and are used worldwide and in all areas of activity. Application development is highly competitive as developers come up with applications with features to meet users' needs [5,6]. Almost all organisations have adopted the new uses offered by these applications. In addition to the development, management of reduced time, developers must deal with the use of increasingly complex platforms.

Indeed, Cueva et al. [7] have shown in their study of embedded multimedia applications that the increase in hardware complexity has led to an increase in application complexity. In addition, all private and confidential informations are easily stored on these applications. This makes the applications prone to vulnerabilities and malicious attacks [8]. Several software vulnerabilities are detected every year. Studies [1,6] show that malware threats are increasing with the continued expansion of the functionality of associated devices. Some applications are harmful and contain vulnerabilities. These applications put users' data at risk. It is therefore necessary to detect vulnerable and malicious software. Protection mechanisms based on signatures (antivirus), firewalls, etc. are sometimes necessary but became ineffective over the years against the new threats from cybercriminals [2–4,9,10]. The constant evolution of threats and malware

attacks that exploit software vulnerabilities requires the use of new techniques to analyze and detect these vulnerabilities. In the remainder of this section, we study these techniques and particularly tracing techniques.

2.2 Tracing

Having efficient and robust applications remains a challenge for developers and researchers. Unfortunately, traditional application protection measures have shown their limits. In the quest to improve the security of software data, researchers have explored solutions based on application traces [11–13]. The purpose of tracing an application is to obtain precise information on its behaviour. Most of this information is dated with precision [11]. The study by Hassan et al. [14] on the traces of web applications shows the possibility of obtaining information such as the date, time, duration and IP address of the web applications used. Zhou et al. [15] add that important information can be collected and transmitted using the TCP/IP protocol. This information makes it possible to link each IP address to the applications running on the device. This shows that the content of the traces is essentially accurate and can be recorded for analysis for security purposes. To obtain this information, the tracing relies on events that will be captured when the system has reached certain states [11,16]. The collection of traces is fluid and only slightly modifies the application's behaviour. The events are produced using trace points (see Sect. 4). Several tracing techniques have been used in previous work. These techniques include debugging, profiling and logging.

In practice, debugging, logging and profiling techniques are similar. All of these techniques are used to present the user with information about the system being monitored so that the user has a better understanding of its behaviour. However, these tools differ in several aspects.

2.3 Debugging, Profiling and Logging Techniques

Debugging. According to Gruber's works [17], debugging is a way of checking the software functions to determine whether it does what it should do. It is used by developers during the validation of an application to discover and eliminate bugs if possible. Debugging therefore has a different purpose than tracing. It allows us to understand the reasons for the errors that occurred during the execution of a program [18]. This is why debugging techniques are used for development purposes. They make it possible to show the user precise and useful messages to understand the causes of system errors. Belkhiri [18] shows in his work that some bugs can remain hidden until integration. Thus, some bugs are difficult to diagnose and solve. Authors of the works [17–20] explain that frameworks with advanced features are needed to debug and optimize applications. One of the limitations of debugging is the step-by-step inspection of the system state. In addition, debugging only identifies bugs that are easy to find and therefore debugged applications can be vulnerable. In addition, debugging is less useful during the execution of an application, so vulnerabilities may escape

it. It is therefore essential to find optimal solutions for diagnosing vulnerabilities and optimising computer programs.

Profiling. As for profiling, it provides users with general informations about the system. For Gruber [17], profiling allows one to verify that the application being analysed is doing what it should be doing at the time it is supposed to. Profiling also provides statistics on memory usage and execution time associated with the monitored system [17,19]. While profiling can identify bottlenecks in applications execution, it doesn't take into account events that occur during the time the system is being monitored. Thus profiling presents to user with averages that correspond to the behaviour that occurred during that time but doesn't take into account the order in which events occur on the system. Furthermore, the number of results presented as a result of profiling an application doesn't increase even if the profiling time increases, unlike tracing. Thus profiling doesn't provide enough information to successfully diagnose problems in an application.

2.4 Logging

Logging makes it possible to record valuable information about the events of a system in the course of its activity, such as access to a system or file, modification or transfer of data [21]. The works [21–23] show that logging resembles tracing in the sense that the aim is also to record events occurring in a system. Also, the data recorded is very high level, unlike the events of tracing which can be very low level. According to these studies, it is noted that the frequency of recording in logging is very low compared to that of tracing, and the information collected is the errors in the program output. Taking into account the new architectures that digital technology is adopting, such as mobile devices, it is difficult for logging alone to solve the problems effectively. On the one hand, it is necessary to manage the interactions between different services and on the other hand to save the path of a trace that passes through various functions. Thus, the logging tools are relatively inefficient for recording frequent events. We will therefore consider tracing tools as optimised logging tools capable of gathering a lot of information.

In sum, debugging, profiling and logging tools do not provide enough information for optimal diagnosis of software vulnerabilities. Tracing that includes these tools is an alternative and inclusive solution. Several tracing tools have been proposed for the visualisation of application traces.

3 Tracing and Visualisation Tools

3.1 Applications Tracing Tools

Tracers are tools responsible for collecting events occurring during system execution and storing them in an orderly fashion in a trace. They used to understand

the functional or interactive behaviour of the system through relevant and valuable data. There are several tracers whose difference lies in the volume of data collected, the associated system and the behaviours represented. These different tracers can save events that occur in the operating system (kernel tracing) or in applications (user-space tracing) within the same trace. Kernel tracing provides enough information about the execution of a system or an application and user-space tracing allows to trace an application by looking at its performance. We are therefore interested in user-space tracing. On the one hand, this tracing allows the modification of the source code of an application to insert tracing points in order to analyze the application's behaviour. On the other hand, because we want to collect more application-specific information. We will study the best user-space tracers based on the performance of each tracer.

Strace. Strace[1] is one of the powerful, simple and easy to use tracers. It provides its user with a record of the system calls that a program makes. This tracer is offered in many distributions and can provide statistics on the time a program spends in the kernel [24]. To obtain a Strace trace, the program must be launched by setting the name of the command to be traced as a parameter to strace. It is possible to specify filters and restrict the trace to certain system calls. One of the advantages of Strace is that it offers the facility to record the opening and closing of files of any program [25]. However, Strace uses only one file per output process and this makes it inefficient. However, Strace uses only one file per process for output, making it very inefficient on parallelized programs, since it uses a lock to allow writing of the collected information. In addition, Strace's performance is worse at tracing system calls, which introduces two extra system calls for each traced system call. This can slow down the execution time of the program.

DTrace. DTrace[2] is a tracer that allows us to run scripts. DTrace has been made compatible with Mac OS X and Linux environments by a team at Oracle [24]. A user can instrument system calls without having to recompile their application code. Since DTrace attaches to probes in the kernel, it is possible to trace all processes running on the system. This tracer is part of Apple Inc.'s development support software.[3]. This is a tool that is known to have no impact on the traced system when trace points are disabled. It uses dynamic instrumentation of user-space applications. DTrace has its own language like C to describe the actions to be taken when a prerequisite is met. This provides a unified interface for user-space application tracing. For performance, DTrace tracing overhead increases when tracing applications due to the number of threads involved in using concurrent execution protection mechanisms when tracing [25,26].

[1] http://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html.
[2] http://dtrace.org/blogs/about/.
[3] https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/.

SystemTap. SystemTap is a tracer more recommended for system administrators to facilitate their access to kernel tracing and retrieve statistics as soon as a problem occurs. It is a tracing tool that combines trace generation and analysis of results. SystemTap allows the use of scripts like DTrace to define actions when a trace point is reached. Scripts can be defined at the beginning or end of functions or in some cases during system calls [26]. The difference is that SystemTap scripts are compiled as native code and run faster than DTrace's bytecode-compiled scripts. With Strace, the difference is that SystemTap analyses the activity of all the processes in the system. SystemTap allows to analyze all interactions of the same type for all processes running on the computer compared to Strace which only analyses one or a few specific processes. This provides more informations for data analysis to identify problems caused by the activity of other processes. However, the synchronisation techniques used by SystemTap mean that the overhead of tracing increases [25]. In addition, during tracing, the calculation of statistics and the flexibility of the tracer are an advantage when it comes to diagnosing performance problems. Unfortunately, the traces are often large and are written to a text file or directly to a console. In addition, the analysis of the results is done during the tracing process, so it is impossible to do more complex or customised analyses afterwards. These reasons show that the use of SystemTap is limited for our study.

LTTng-UST. Linux Trace Toolkit Next Generation User-Space Tracer (LTTng-UST) is the component of the LTTng[4] kernel tracer that runs entirely in user mode. This tracer has been designed to minimize its impact on the traced system. It uses the same architecture as LTTng and relies on trace points inserted at arbitrary locations in the code, which will send informations about the state of the system to the tracer when encountered during execution of the instrumented program. Thus, the program is dynamically linked during its execution to the liblttng-ust library. To ensure scalability to multiple cores and to be efficient, this tracer allocates a circular buffer for each execution core, allowing to instrument programs with parallelized execution without penalizing the different execution threads [24]. In addition, the buffers are managed through atomic operations and are shared with a background process responsible for transferring the stored data to disk or sending it across the network, and flushing the buffers before they are filled. However, the buffers can fill up too quickly, if the network or disk throughput for example is lower than the event generation rate. New events overwrite older ones. A buffer backup can be triggered if necessary. For example when an error occurs or when too much latency is detected, a trace of the few seconds preceding the incident becomes available for analysis. By default, LTTng accepts to lose events rather than blocking the process waiting for the disk or the network, in order not to slow down the execution of the program. The advantage is therefore to reduce system disruption and avoid having to manage the storage of large traces. With these different developed modules, LTTng-UST is able to instrument programs written in C/C++, Java and even Python. Furthermore,

[4] https://lttng.org.

it is easy to combine traces produced in user-space by LTTng-UST and kernel traces produced by LTTng. Tracing with LTTng-UST doesn't require any system calls, which makes it very powerful. Events are written by applications to buffers in shared memory. A single daemon is responsible for collecting events generated by all instrumented applications on the system.

3.2 Viewing Traces

The virtualization tools provide appropriate views of the execution of programs in order to understand the information gathered. The raw trace data can be displayed, but when the traced program becomes complex, it is more difficult to get an overview of the application's execution in order to detect functionality or performance problems. Several trace visualisation tools have been proposed to analyse events. In this study, we favour the tools for visualising traces generated in CTF format, although they are binary and therefore difficult to read directly.

Babeltrace. Babeltrace is a LTTng trace visualization tool used to print the content of a trace to the console. This tool makes it easy to read the traces without performing any analysis. In addition to the provided libbabeltrace library that allows other programs to read LTTng traces, it allows to display CTF traces as ordered textual events. These displayed events are precisely stamped with the name of the event and the arguments recorded at the time the trace point was encountered at system execution. The advantage of this tool is that it can convert traces from CTF format to another format and vice versa while providing a programming interface in C or Python language. This makes it possible to filter the desired events for analysis. Also, the programming interfaces allow the conversion of binary data into data structures that can be modified and used by other algorithms. The weakness of Babeltrace lies in the slow reading of traces in CTF format in case the number of events is high.

Trace Compass. Formerly known as the Tracing and Monitoring Framework plugin within the Eclipse environment, Trace Compass is a popular trace visualization and analysis tool and supports several trace formats including the CTF format. It is designed to quickly process very large traces and process events based on finite state machines to present different results. This tool provides clear and differentiated views from recorded traces. Also, the conversion of traces in CTF format to a human-readable format is done using state machines. These state machines are very efficient for saving large numbers of recorded events [25], and the construction of the state tree is particularly optimised. Thus, when a trace is loaded into the software, the state tree is calculated according to the interval visible in the current view, which allows this tool to display traces quickly, and users to be able to explore the different recorded events efficiently. In addition, various views have been developed to analyse the traced system and display statistics relating to the system during the period it was traced. To obtain richer views, it is possible to add Java code to the project or even attach

scripts written in Python to process the visualized traces without modifying the source code. Trace Compass is therefore a preferred tool for visualizing recorded traces and for conducting fast and powerful analyses. A limitation of Trace compass is that each analysis is confined to its own view. It is therefore necessary to combine the information that each analysis offers about the same resource.

4 Approach

4.1 Choice of Applications

In the rest of our work, we have restricted our study to mobile applications, particularly Android applications. Our choice is motivated by the popularity of Android ([8,27]), the growing number of vulnerabilities in applications (Gartner report[5], [27]), the risks of threats to which users are exposed ([15,28]) and the inadequacy of the solutions for the protection of the data that transit these applications ([1,9,29]). In fact, functions that were previously reserved for hardware are now taken into account by mobile applications. In addition, the openness of Android allows developers to take advantage and the fact that applications are free to download and use has helped attract users [8,15]. According to Gartner's Digital Report, Android is in first place with over 82% of the mobile market and over 2 billion shipments by 2021. This expansion of applications has also created challenges related to data security. Several studies show that vulnerabilities in applications make it the most vulnerable system and the most targeted by malicious attacks [2,4,8,27,30]. The scale of attacks targeting Android users is considerable. These attacks exploit vulnerabilities, the increasing number of which has caused enormous damage to users. Unfortunately the threat of malware is growing every year and new malware such as ransomware has evolved in sophistication to evade existing scanning techniques. The complexity of malware, its evolution, the increase in damage caused by its attacks and the inadequacies of traditional protection mechanisms prompt us to explore execution traces to improve the protection of application data in this system. In addition, few works have focused on the tracing of mobile applications.

4.2 Using Machine Learning Techniques

In view of the number of vulnerabilities in Android applications, we propose in our approach to combine machine learning techniques with tracing for vulnerability detection. Machine learning is one of the new approaches used to detect vulnerabilities in Android applications [30,31]. It has algorithms that combined with our approach will improve the identification of software vulnerability risks in Android and thus reduce the threat of malware. After the recovery of events and the visualization of the obtained traces we will extract the characteristics in these traces to build vectors suitable for the analysis with the algorithms.

[5] https://www.gartner.com/en/information-technology/insights/top-technology-trends/top-technology-trends-ebook.

In addition, we will build a dataset to train our model to learn from the data. To the vectors we will apply Logistic Regression (LR), Linear Discriminant Analysis (LDA), K-Nearest Neighbors (KNN), Decision Tree Classifier (DTREE-CART), Naive Bayes (NB), MLP Classifier (MLP), Random Forest Classifier (RFOREST) and Support Vector Machine (SVM). Then we will compare the results of these algorithms on vulnerability detection from execution traces. In this way we will obtain results that improve the accuracy, efficiency and effectiveness of vulnerability detection.

4.3 Model Proposed

In the model we propose in Fig. 1, our approach is an extension of tracing techniques to Android applications to which we combine machine learning techniques. This model has two stages of analysis of Android applications. In the implementation of our Android applications tracing approach, we will collect applications (both malware and benign) from several sources to instrument the code.

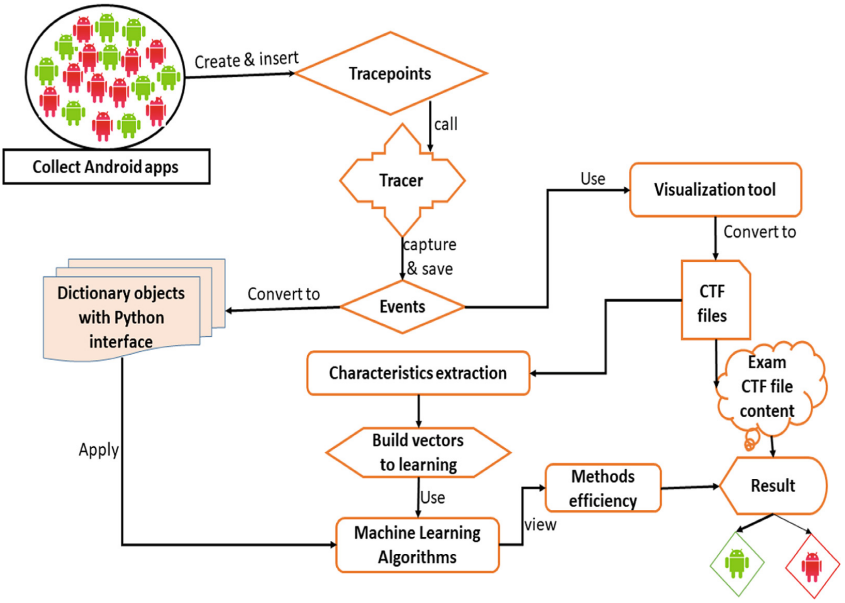

Fig. 1. Model using Android application execution traces

The first step is to use tracepoints. These tracepoints will function to make a call containing information about the state of the application. With the insertion of tracepoints, we will have data on the operation of the traced application and also a description of its interactions. To capture the events that are the

data exported by the tracepoints, we reserved the LTTng-UST tracing tool. This tracer is known for its low overhead and its ability to trace applications running in the user-space. The collected events are stored in a trace in an orderly fashion. Then, we will use the visualization tool Trace Compass to convert the collected binary data into a CTF file. Finally, as this file is readable, we will analyse its content to identify possible precursor behaviours of attacks according to functional and interactive gaps.

Tracing and visualisation tools can sometimes provide an effective first approach to tracing but are not sufficient on their own to identify vulnerabilities. This encourages us to combine learning techniques to refine the examination of the traces. The second step is therefore the extraction of features from the CTF files. These features will be used to construct eigenvectors for analysis with the algorithms seen previously. The aim is to convert the data into vectors of numbers for model learning so that it can be used as a basis for identifying the origin of undesirable application behaviour. Using the Python interface, the events in the trace can be read and converted into dictionary objects. These objects, after various modification steps, can also be used to feed machine learning algorithms.

5 Conclusion

New threats against digital suggest new approaches to improve data protection. In this paper we have presented a model to improve the vulnerabilities detection in Android applications. Our model combines tracing and machine learning techniques to analyze Android applications execution traces. These traces have proven to be valuable in analysing applications through a comparison of functionality with its runtime behaviour. For this purpose, we selected LTTng-UST as tracer to capture and record events and Trace compass as visualization tool. To these powerful tools, we proposed to associate machine learning algorithms which allows to have detailed informations on the application behaviour but also on the trace structure.

The main contribution of this paper is the offering of a model that combines user-space tracing techniques with machine learning techniques for advanced analysis of Android applications. In addition, this paper provides a comparative study of tracing techniques, tracers and also runtime trace visualization tools. Our approach is to limit the actions of malware that use evasion techniques. And, this will significantly increase the detection of vulnerabilities and also real problems in Android applications. In the future, we plan to compare the results obtained by applying our model with other approaches to evaluate the success rate and effectiveness of our model. The objective is to have applications that are resistant to malware evasion techniques.

References

1. Nguyen, K.D.T., Tuan, T.M., Le, S.H., Viet, A.P., Ogawa, M., Minh, N.L.: Comparison of three deep learning-based approaches for IoT malware detection. In: 10th International Conference on Knowledge and Systems Engineering (2018)

2. Dehkordy, D.T., Rasoolzadegan, A.: A new machine learning-based method for android malware detection on imbalanced dataset. Multimedia Tools Appl. **80**(16), 24533–24554 (2021). https://doi.org/10.1007/s11042-021-10647-z
3. Lin, G., Wen, S., Han, Q-L., Zhang, J., Xiang, Y.: Software vulnerability detection using deep neural networks: a survey. In: Proceedings of the IEEE (2020). https://doi.org/10.1109/JPROC.2020.2993293
4. Dong, S., et al.: Understanding android obfuscation techniques: a large-scale investigation in the wild. In: Beyah, R., Chang, B., Li, Y., Zhu, S. (eds.) SecureComm 2018. LNICST, vol. 254, pp. 172–192. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01701-9_10
5. Garcia, J., Hammad, M., Malek, S.: Lightweight, obfuscation-resilient detection and family identification of android malware. ACM Trans. Softw. Eng. Meth. (TOSEM) (2018)
6. https://www.gartner.com/en/information-technology/insights/top-technology-trends/top-technology-trends-ebook
7. Cueva, P.L., Bertaux, A., Termier, A., Méhaut, J.F., Santana, M.: Debugging embedded multimedia application traces through periodic pattern mining. In: Proceedings of the Tenth ACM International Conference on Embedded Software, EMSOFT 2012, pp. 13–22 (2012). https://doi.org/10.1145/2380356.2380366
8. Koala, G., Bassolé, D., Zerbo/Sabané, A., Bissyandé, T.F., Sié, O.: Analysis of the impact of permissions on the vulnerability of mobile applications. In: International Conference on e-Infrastructure and e-Services for Developing Countries, AFRICOMM 2019, pp 3–14 (2019). https://doi.org/10.1007/978-3-030-41593-8_1
9. Ghaffarian, S.M., Shahriari, H.R.: Software vulnerability analysis and discovery using machine-learning and data-mining techniques: a survey. ACM Comput. Surv. **50**(4), 1–36 (2017)
10. Lei, T., Qin, Z., Wang, Z., Li, Q., Ye, D.: Evedroid: event-aware android malware detection against model degrading for IoT devices. IEEE Internet Things J. (2019). https://doi.org/10.1109/JIOT.2019.2909745
11. Lebis, A.: Capitaliser les processus d'analyse de traces d'apprentissage : modélisation ontologique et assistance à la réutilisation", Thèse, Sorbonne Université (2020). https://tel.archives-ouvertes.fr/tel-02164400v2
12. Galli, T., Chiclana, F., Siewe, F.: Quality properties of execution tracing, an empirical study. Appl. Syst. Innov. **4**, 20 (2021). https://doi.org/10.3390/asi4010020
13. Hojaji, F., Mayerhofer, T., Zamani, B., Hamou-Lhadj, A., Bousse, E.: Model execution tracing: a systematic mapping study. Softw. Syst. Model. **18**(6), 3461–3485 (2019). https://doi.org/10.1007/s10270-019-00724-1
14. Hassan, N.A., Hijazi, R.: Digital Privacy and Security Using Windows, CA Apress, Berkeley (2017). https://doi.org/10.1007/978-1-4842-2799-2
15. Zhou, D., Yan, Z., Fu, Y., Yao, Z.: A survey on network data collection. J. Netw. Comput. Appl. **116**, 9–23 (2018). https://doi.org/10.1016/j.jnca.2018.05.004
16. Lazar, J., Feng, J.H., Hochheiser, H.: Chapter 12 - Automated Data Collection Methods. Research Methods in Human Computer Interaction, 2nd edition, Elsevier, Britain, pp 329–368 (2017). https://doi.org/10.1016/B978-0-12-805390-4.00012-1
17. Gruber, F.: Performance debugging toolbox for binaries: sensitivity analysis and dependence profiling. pp 3–10 (2020). https://tel.archives-ouvertes.fr/tel-02908498
18. Belkhiri, A.: Analyse de performances des réseaux programmables, à partir d'une trace d'exécution (2021). https://publications.polymtl.ca/9988/1/2021_AdelBelkhiri.pdf

19. Venturi, H.: Le débogage de code optimisé dans le contexte des systèmes embarqués", pp. 13–40 (2008)
20. Iegorov, O.: Data mining approach to temporal debugging of embedded streaming applications, pp 89–95 (2018). https://tel.archives-ouvertes.fr/tel-01690719
21. Bationo, Y.J.: Analyse de performance des plateformes infonuagiques, École Polytechnique de Montréal, pp. 19–28 (2016)
22. Reumont-Locke, F.: Méthodes efficaces de parallélisation de l'analyse de traces noyau (2015). https://publications.polymtl.ca/1899/1/2015_FabienReumontLocke.pdf
23. Ravanello, A.: Modeling end user performance perspective for cloud computing systems using data center logs from big data technology. Thesis (2017)
24. Kouamé, K.G., Ezzati-Jivan, N., Dagenais, M.R.: A flexible datadriven approach for execution trace filtering. In: IEEE International Congress on Big Data (BigData Congress: New York, NY, USA (2015). https://doi.org/10.1109/bigdatacongress.2015.112
25. Bationo, Y.J., Ezzati-Jivan, N., Dagenais, M.R.: Efficient cloud tracing: from very high level to very low level. In: IEEE International Conference on Consumer Electronics (ICCE 2018), Las Vegas, NV, USA (2018). https://doi.org/10.1109/icce.2018.8326353
26. Ezzati-Jivan, N., Bastien, G., Dagenais, M.R.: High latency cause detection using multilevel dynamic analysis. In: Annual IEEE International Systems Conference SysCon: Vancouver. Canada (2018). https://doi.org/10.1109/syscon.2018.8369613
27. Agrawal, P., Trivedi, B.: A survey on android malware and their detection techniques. In: IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT) (2019). https://doi.org/10.1109/ICECCT.2019.8868951
28. Qamar, A., Karim, A., Chang, V.: Mobile malware attacks: review, taxonomy and future directions. Future Gener. Comput. Syst. **97**, 887–909 (2019). https://doi.org/10.1016/j.future.2019.03.007
29. Zhou, Q., Feng, F., Shen, Z., Zhou, R., Hsieh, M.-Y., Li, K.-C.: A novel approach for mobile malware classification and detection in Android systems. Multimedia Tools Appl. **78**(3), 3529–3552 (2018). https://doi.org/10.1007/s11042-018-6498-z
30. Sestili, C.D., Snavely, W.S., VanHoudnos, N.M.: Towards security defect prediction with AI (2018). arXiv:1808.09897. http://arxiv.org/abs/1808.09897
31. Fernández, A., García, S., Galar, M., Prati, R.C., Krawczyk, B., Herrera, F.: Imbalanced classification for big data. In: Learning from Imbalanced Data Sets, pp. 327–349. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98074-4_13