Software Vulnerabilities' Detection by Analysing Application Execution Traces

Gouayon Koala[1], Didier Bassolé[2], Telesphore Tiendrebeogo[3], Oumarou Sié[4]
Laboratoire de Mathématiques et d'Informatique, Université Joseph Ki-Zerbo
Ouagadougou, Burkina Faso[1,2,4]
Laboratoire d'Algèbre, de Mathématiques Discrètes et d'Informatique
Université Nazi Boni, Bobo-Dioulasso, Burkina Faso[3]

***Abstract*—Over the years, digital traces have proven to be significant for analyzing IT systems, including applications. With the persistent threats arising from the widespread proliferation of malware and the evasive techniques employed by cybercriminals, researchers and application vendors alike are concerned about finding effective solutions. In this article, we assess a hybrid approach to detecting software vulnerabilities based on analyzing traces of application execution. To accomplish this, we initially extract permissions and features from manifest files. Subsequently, we employ a tracer to extract events from each running application, utilizing a set of elements that indicate the behavior of the application. These events are then recorded in a trace. We convert these traces into features that can be utilized by machine learning algorithms. Finally, to identify vulnerable applications, we train these features using six machine learning algorithms (KNN, Random Forest, SVM, Naive Bayes, Decision Tree-CART, and MLP). The selection of these algorithms is based on the outcomes of several preliminary experiments. Our results indicate that the SVM algorithm produces the best performance, followed by Random Forest, achieving an accuracy of 98% for malware detection and 96% for benign applications. These findings demonstrate the relevance and utility of analyzing real application behavior through event analysis.**

***Keywords*—Execution traces; events; vulnerability detection; malware; applications**

I. Introduction

The prevalence of malicious applications has significantly increased in recent years. Unfortunately, as digital technology continues to advance, the number of vulnerabilities in applications is also growing exponentially, thereby leaving users even more vulnerable. Since these applications handle highly personal and sensitive data, it remains a significant challenge for researchers and application providers to find effective and efficient solutions. Despite the efforts described in the existing literature to safeguard data, the threat remains very real. Moreover, in recent years, it has become even more severe as cybercriminals increasingly employ evasion techniques to bypass existing protection measures [1], [2]. Not only are the majority of available solutions limited or inadequate against the sophisticated tactics of cybercriminals, but these malicious actors are also becoming more organized and motivated [2], [3], [4], [5]. Consequently, ensuring data security and protection has become an essential and urgent concern. It is crucial, therefore, to urgently discover solutions that can minimize the exploitation of software vulnerabilities and mitigate the risk of attacks targeting user data [3], [6].

Among the techniques employed to detect malware in recent years, machine learning has been utilized [7]. This is associated with the static approach ([8], [9]) or the dynamic approach ([1], [10], [11]), depending on the methods employed. In the literature, the hybrid approach is increasingly being utilized to leverage the advantages of both the static and dynamic approaches, thus partially mitigating the limitations inherent in each of these two approaches. As our approach involves utilizing data from the Android's manifest file for static analysis and execution traces for dynamic analysis, it can be categorized as a hybrid approach. By combining the analysis of application execution traces with machine learning techniques, we aim to enhance malware detection. Previous studies have emphasized the significance and utility of traces in monitoring computer system behavior [11], [12], [13], [14], [15].

The collected traces enable us to comprehend the functioning of a system and identify anomalies, deviations in operation, suspicious behavior, and more. Hence, traces contain pertinent and valuable information for analyzing the behavior of systems in general, as well as applications specifically during their execution. Although traces are beneficial, they are less commonly utilized for identifying malicious applications. Instead, they are typically employed for debugging, profiling, or logging purposes. This study aims to assess the hypothesis that traces of application execution are high-quality data that can be used to analyze and detect malicious applications [16]. Consequently, the solution proposed in this study is founded on capturing relevant behavioral elements (events) during application execution, based on pertinent characteristics. These events are recorded in the traces. Subsequently, the values of the behavioral features are extracted in the form of dictionary objects or converted into eigenvectors using appropriate tools for analysis with machine learning algorithms. The primary objective of this study is to effectively detect malware in order to enhance the safeguarding of private data transmitted through applications. Therefore, it presents a proactive solution that diminishes cybercriminals' attack vectors. The experimental results demonstrate the significance of execution traces in identifying software vulnerabilities. This study contributes to malware detection in the following ways:

- We present a model that effectively and efficiently identifies malware by utilizing a blend of static features (permissions and characteristics) and behavioral features (traces). The features we have selected allow us to describe the dynamic behavior of applications. Furthermore, these features are comprehensive, en-

compassing attributes extracted from the Android-Manifest file as well as features extracted during application execution.

- We have extracted five (05) relevant features to characterise the behaviour of Android applications. The numerical values of these features vary from one application to another. We use six (06) classifiers, namely Support Vector Machine (SVM), k-Nearest Neighbor (kNN), Random Forest (RF), NB, MLP and DTREE-CART, to identify malware. We compare the detection performance of these different classifiers.
- We conducted analyses on a dataset containing 8014 traces from benign and malware applications collected from Google Play (15%) and Drebin (85%). Experimental results show the effectiveness of the model with a detection accuracy of more than 98% with the SVM algorithm.

The rest of the document is structured as follows: the Section II deals with some previous studies and research into traces and vulnerability detection methods. In Section III we detail the process of collecting traces and converting them into features through trace generation, data pre-processing and feature vector formation. Section IV presents the construction of the data set and experimental setup. In addition, the results obtained will be presented in this section. We conclude the work in Section V.

II. Related Work

Over the last few years, the digital world has seen an impressive development in malicious software, which represents a major threat [3], [4]. The consequences of this malware for users are enormous. On an ongoing basis, a number of researchers have proposed methods and techniques for detecting malware in applications [7], [8], [17], [18], [10]. The aim is to improve data protection methods by reducing threats and attacks against private data. Unfortunately, their efforts are coming up against determined cybercriminals who are more innovative in their malicious behaviour [3], [2], [19], [5], [17]. They are increasingly using sophisticated techniques to bypass security solutions or evade the control systems in place. It is therefore crucial and urgent to find effective solutions to protect data. Several methods and techniques based on static and dynamic approaches are proposed [7], [19], [13], [16]. Previous works [4], [19], [16] have proposed literature reviews on both analysis approaches and their weaknesses [3], [17], on analysis techniques, [7], [20], on the use of machine learning techniques [7], [8], [17], on digital traces [11], [16]. Nevertheless, we will mention some of the work related to traces, especially as our approach is a hybrid one.

In static analysis, the source code is examined and representative features (libraries, opcodes, API calls, permissions, function calls, etc.) are extracted. In contrast to the static approach, for dynamic analysis, representative features are extracted during the execution of the application by monitoring its behaviour. Features such as system calls, file behaviour (access, create, read, modify, delete), registry access (create and modify), network traffic, are relevant data used by some authors to study application behaviour and detect malicious actions.

The authors of the works [8], [9], [10], [20], combined machine learning techniques with one or other of these approaches, depending on their methodology to improve detection. The authors Al-Hashmi et al. [10] selected several features from the extracted features and combined them with different machine learning techniques to train a model. The results obtained are encouraging with their DeepEnsemble model. Numerous other works on detecting malware in Android applications extract certain information to distinguish malicious applications from benign ones. This is the case of the work by Bassole et al. [18] and the authors [8] and [9]. These authors extracted authorisations and other functionalities as features, and combined them with machine learning methods to detect malware.

As for traces, they were used by the authors [13], [15], [14], [21], [22] and [23] in their work. The authors of the references [24], [25] [26] and [27] used traces to detect their code errors through debugging. In the studies [24] and [26], traces are used for profiling while the authors [28], [29] and [30] use them as a means of studying logging. All these techniques using traces (debugging, profiling and logging) do not provide enough information for optimal diagnosis of flaws in applications. It was in the web and network domain that the first uses of traces were useful. Hassan et al. [21] used traces to detect vulnerabilities in web sites and Zhou et al. [22], used them to analyse anomalies in TCP/IP networks. Several other studies show that traces provide more relevant results when combined with machine learning techniques [31], [32]. Studies such as that carried out by Razagallah et al. [11] show us that traces are an invaluable source of information on the execution behaviour of a program. This information can be used to detect malicious software in Android applications. The authors have therefore built up a dataset based on traces that can be exploited according to research needs.

Despite all these malware detection methods and protection measures, cybercriminals often manage to escape and exploit vulnerabilities with more sophisticated attacks. Between the repackaging of certain benign applications for malicious purposes, new families of malware, vulnerabilities (known and unknown) and inadequacies in data protection, there is a need to explore possible solutions to limit the risk of attacks.

With the ever-changing and innovative nature of malware, detection based solely on one approach or type of functionality cannot meet data protection needs. In this study, we therefore evaluate an analysis model that uses events collected during application execution to analyse the malicious behaviour of these applications (Fig. 1). This is a hybrid approach that takes advantage of both static and dynamic approaches and combines machine learning techniques. The traces generated contain sensitive information extracted and trained by the algorithms for detecting software vulnerabilities. In this way, static characteristics (permissions and features) are extracted and dynamic characteristics are captured in order to obtain data that is more relevant and better suited to improving malware detection performance.

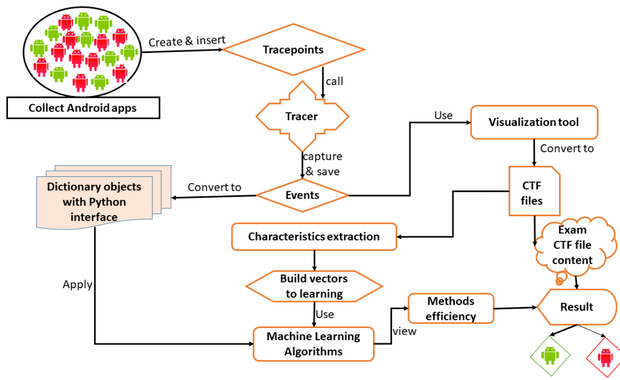

Fig. 1. Model using android application execution traces.

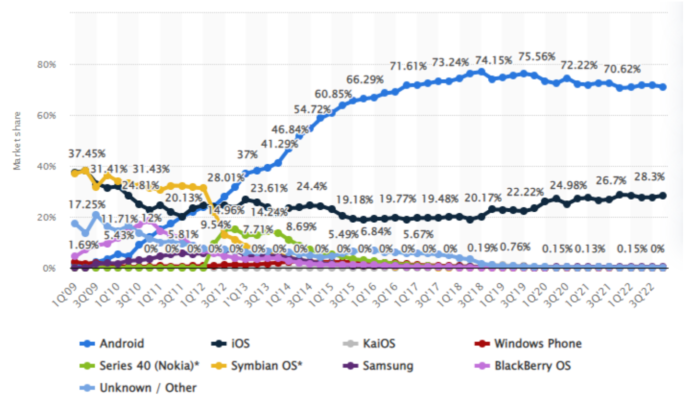

Fig. 2. Growth rate of android applications compared with other applications from 2009 to 2022.

III. Trace Collection and Processing

In this section, we present the process for extracting features and events, the construction of our dataset, and the features selected for training the machine learning algorithms. We also present the tools and equipment used to collect the events.

A. Choice of Android Applications

To evaluate our model, we used execution traces generated from Android applications. Our choice is motivated by the fact that the high number of these applications with a share of over 82% of mobile applications, according to Gartner's 2021 report[1]. According to this report more than 2 billion will be delivered in 2021. All this popularity (see Fig. 2) combined with Android's security model makes users of these applications a prime target for malware writers. The scale of attacks targeting Android users is considerable [1].

B. Behavioural Data Extraction (Events)

Application behaviour, operations performed and system performance are among the essential and useful data provided by application execution traces. This data can be used for analysis, debugging, performance optimisation, problem detection and many other tasks related to application monitoring and diagnosis. For analysis, the features to be extracted from the events depend on the context of the application and the information you wish to use to construct the eigenvectors. In this study we have specified the behavioural features to be extracted in the features_to_extract list (Table I). These features are relevant, we believe, to understanding the actual behaviour of the application being run. Several events are captured and recorded in a file (the trace). All events are generated with the LTTng tracer. Also, to extract characteristics during the execution of each application, we created a dynamic environment with the Genymotion emulator version 3.3.3 with a Google Nexus 5 API 11 device. Each of the applications is installed and then run. In the Algorithm 1, we describe the process of collecting events, the building blocks of traces.

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

TABLE I. List of Behavioural Data to be Extracted

Event data	Type of data	Description
Timestamp	Numeric	This is the timestamp that indicates the precise moment when each event occurred. It is useful for temporal analysis of events and for understanding the chronological order in which they occurred
"Name"	String	This is the name of the event, representing the type or category of the event. It can indicate a specific action performed by the application, a function call or a system operation
PID (Process ID)	Numeric	This is the process identifier (PID), which is a unique number assigned to each process running on the system. It identifies the process that caused the event
TID (Thread ID)	Numeric	This is the thread identifier (TID), which is a unique number assigned to each thread in a process. It identifies the specific thread at the origin of the event
"Syscall" (System Call)	String	This is the set of data that represents a request from the running program to the operating system kernel to perform a specific operation. For example, read or write data, access external resources, create files, allocate memory, etc. The "syscall" field indicates the specific system call associated with the event
Retval (Return Value)	Numeric	This represents the return value of the system call (syscall). It is a numerical value that indicates the result of the operation performed by the system call, such as the success of an operation or the occurrence of an error
Duration	Numeric	Duration represents the time elapsed between the start and end of an event. It is used to measure the execution time of each specific operation

C. Data Pre-processing

Once the traces have been generated, the data collected must be made useful for the rest of the process. This stage is essential and requires appropriate tools to transform behavioural data into features. It is this phase that produces data that can be used by machine learning algorithms. During the pre-processing phase, only data that can contribute to improving detection or classification is retained from the data collected. This data should maximise the accuracy of the results obtained. Unnecessary data is therefore ignored. Given that the data we collected in the previous stage is unstructured data, it comes in different formats and is sometimes unreadable. Also, they generally contain redundant and unnecessary features, with missing values, symbols, punctuation and spaces. To prevent unnecessary data from negatively influencing the results, we converted the traces into Python dictionary objects. The Algo-

Algorithm 1: Extracting Events from an Application

```
Entry :
        Application : .apk
        events_to_extract[]: contains the list of
    elements to be extracted
Output:
        T: list of traces (content and metadata)
1  Load application into memory //Specify application
     package name
2  package_name = "MyApp"
3  source_code = decompile(app.apk)//Decompile the
     apk to obtain the source code
4  trace_code = insert_tracepoints(source_code)
5  //Insert tracepoints in the source code to capture the
     desired events (calls to special functions or macros
     that record events)
6  Run the application several times //(2 to 5 times)
7  Run Tracer //Configure the lttng-ust tracing tool to
     collect events
8  events = extract(trace_code) // collect events
     generated on the state of variables, function calls,
     errors, system events, etc.
9  T=[] // Initialise the list of traces
10 foreach event ∈ events do
11 |  if event ∈ events_to_extract then
12 |  |  // for an event element in the list
   |  |    events_to_extract
13 |  |  trace = event // Create a new trace with the
   |  |    event
14 |  |  T.append(trace) // Add the track to the list of
   |  |    tracks
15 |  end
16 |  else
17 |  |  Continue with the next event
18 |  end
19 end
20 Return T
```

Algorithm 2: Convert Generated Traces into Python Dictionary Objects

```
Entry :
        trace_file : events file
Output:
        event_dicts : events converted into dictionary
    objects
1  Define the path to the directory containing the traces
2  Define the name of the traces session
3  events_to_extract = [timestamp, "name",
     pid,tid,"syscall", retval,duration, cpu]
4  Run TraceCompass // to convert evenements
5  Load the file containing extracted events
6  events = open("trace_file", "read") //Storing events in
     an events variable
7  event_dicts = { } // Initialise the object dictionary
8  foreach event ∈ events do
9  |  if event["name"] ∈ events_to_extract then
10 |  |  //If the event name is in the list
   |  |    events_to_extract
11 |  |  event_dict = {//create a dictionary event_dict
   |  |    with the variables
12 |  |  timestamp: event[timestamp],
13 |  |  "name": event["name"],
14 |  |  pid: event["fields"][pid],
15 |  |  tid: event["fields"][tid],
16 |  |  "syscall": event["fields"]["syscall"],
17 |  |  retval: event["fields"][retval],
18 |  |  duration: event[duration],
19 |  |  cpu: event[cpu]
20 |  |  }
21 |  end
22 |  else
23 |  |  continue with the next event
24 |  end
25 |  event_dicts.append(event_dict) //Adding the
   |    dictionary to the list
26 end
27 return event_dicts
```

rithm **2** presents this transformation process. Once converted, machine learning algorithms use these dictionaries to identify malware from benign software. We also use the Trace compass visualisation tool to convert the traces into CTF files. As the contents of these files are readable, they are used to construct feature vectors, as indicated by the algorithm **3**. In this way, machine learning algorithms can use these feature vectors to detect vulnerable applications, and therefore behaviours that are precursors to possible attacks.

D. Features Representation

Feature extraction creates new feature sets in which the typical malware example is better represented than the use of the original features. This feature-derived data extracted from software behavioural data improves the accuracy of malware detection. Before extracting these characteristics from the traces, we statically extracted the permissions and features of each application. This brings the total number of feature fields to five (05) for vulnerability analysis. For the detection of a vulnerable or malicious application by the machine learning model, the analysis focuses on the features taken from the execution traces and the AndroidManifest file. These fields include all the important information from the traces. These features are based on:

- C1(Searching for abnormal activity): The aim is to analyse the values in this field to identify any activity that does not conform to the expected behaviour of the application. This includes inappropriate access to system resources, attempts to modify critical files and suspicious communications with external servers. The application will be detected as vulnerable.
- C2 (Error and exception detection): This involves identifying errors and exceptions reported in the execution trace and contained in this field. If there are frequent errors in the traces, such as access violations or security exceptions, this indicates potential vulnerabilities in the application.
- C3(Verification of privileges): this field contains the values of the analysis of system calls made in the trace

Algorithm 3: Transformation of Events into CTF Files to Construct Eigenvectors

Entry :
  trace_file : input trace file
  features_to_extract : list of characteristics to be extracted from events
Output:
  feature_vectors: list of eigenvectors constructed from events
1 Load traces // with Trace Compass from the file trace_file
2 Configuring CTF conversion parameters
3 Convert tracks to CTF // format using Trace Compass
4 Loading converted CTF files
5 Initialise feature_vectors as an empty list
6 **foreach** *event belonging to the converted CTF files* **do**
7   Extract the characteristics specified in features_to_extract to converted CTF files
8   Construct a feature vector for the event using the extracted values
9   Add the feature vector to the list feature_vectors
10 **end**
11 Return feature_vectors

and checks whether they are appropriate for the application in question. For example, system calls relating to access to files, processes or network resources may reveal unauthorised access attempts.

- C4(Searching for suspicious network behaviour): This involves examining the values in this field, which represent network communication activities in the trace. If there are suspicious outgoing connections to unknown IP addresses or domains, unauthorised protocols or unencrypted transmissions of sensitive data, then the application is considered vulnerable.
- C5(Detecting malicious behaviour): This involves comparing the permissions and features fields in the Android's manifest file with the *authorisations.txt* and *features.txt* lists. These lists contain all the permissions and features declared in the official Android documentation.

Representing the functionalities represented by each field (C1, C2, C3, C4, C5) in figures means that the counter values can be incremented if a suspect element is present. These values are used to create the data set. The final value of the counter is compared with its initial value. If the final value is equal to the initial value, this implies normal behaviour and therefore a benign application. Otherwise, for any other value different from the initial value, the model assumes that the application is vulnerable.

IV. Implementation and Results

A. Dataset and Experimental Setup

1) Dataset: To experiment with our approach, we collected a number of Android applications that included both benign and malicious apps. We acquired benign apps from Google Play[2] and malicious apps from Drebin[3] and built a dataset of 8014 traces (benign apps and malicious apps). This approach gives us apk's that have undergone Google's verification tests before being published on its site. Nevertheless, all applications are downloaded and then analysed on VirusTotal[4] with a considerable number of antivirus software for detection. The results of these scans are used to group the applications into benign and malware. Applications are selected according to several criteria. For benign applications, almost all sectors of activity are taken into account (tourism, news, health, education, financial transactions, justice, culture, religion, job search, history, geolocation and entertainment, etc). Several types of malware were collected (repackaging, privileges, sending sms, stealing information, advertising, etc.). We formed a set of 8014 traces for analysis.

2) Experimental setup: We conducted our experiments on a computer Inter(R), Core(TM) i3-4160, CPU @ 3.60GHzx4 with with 12GB RAM running on Ubuntu 22.04.2 LTS 64 bits and GNome 42.5. We have installed the LTTng 2.13 plotter including LTTng-tools[5]2.13.9, LTTng-UST[6]2.13.5 and LTTng-modules[7]2.13.9. The models are built with Python 3.10.6 and GCC 11.3.0.

Evaluation metric: The metrics precision (P), recall (R) and F-measure (F1) , Accuracy are proposed to evaluate our method. The precision, recall and F1 for example are defined as:

$$P(precision) = \frac{TP}{TP+FP} \qquad R(recall) = \frac{TP}{TP+FN}$$

$$F1 = \frac{2*P*R}{P+R} \qquad Accuracy = \frac{TP+TN}{TP+FP+FN+TN}$$

Where :

- TP (True Positive): when the actual class and the predicted class are all yes.
- TN(True Negative): when the actual class and the predicted class are all no.
- FP(False Positive): when the actual class is no and the predicted class is yes.
- FN(False Negative): when the actual class is yes and the predicted class is no.

B. Results

To analyse the performance of our model, we used six (06) machine learning algorithms including the K-Nearest Neighbors classifier(KNN), the Decision Tree Classifier (DTREE-CART), Naive Bayes (NB), MLP classifier, Random Forest (RFOREST) classifier, Support Vector Machine (SVM) which were selected on the basis of the results of several preliminary experiments carried out.

[2]https://play.google.com/
[3]https://www.sec.cs.tu-bs.de/~danarp/drebin/download.html
[4]https://www.virustotal.com/
[5]git://git.lttng.org/lttng-tools.git
[6]git://git.lttng.org/lttng-ust.git
[7]git://git.lttng.org/lttng-modules.git

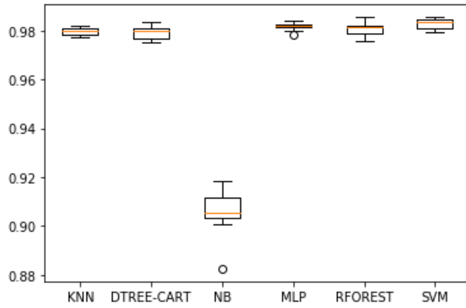

Fig. 3. The results obtained with each algorithm.

The Table II presents a comparison of the performance of the proposed approach with the precision, recall and F1 score measures for detecting software vulnerabilities on Android. The results show good performance for all the machine learning algorithms used. The best performance is given by the SVM algorithm with an F-score of 0.99 for malware detection and 0.89 for benign software detection.

TABLE II. Comparison of ML Algorithms

	Malware			**Benign apps**		
	Precision	***Recall***	***F1-score***	***Precision***	***Recall***	***F1-score***
KNN	0.98	0.99	0.99	0.96	0.75	0.84
DTREE-CART	0.98	0.99	0.99	0.93	0.80	0.86
NB	0.91	0.96	0.95	0.31	0.24	0.27
MLP	0.98	0.99	0.99	0.96	0.77	0.85
SVM	**0.98**	**1.00**	**0.99**	**0.96**	**0.77**	**0.89**
RFOREST	0.98	1.00	0.99	0.94	0.81	0.87

TABLE III. Implementation for All ML Algorithms

	Macro Precision	**Macro Recall**	**Macro F1-score**	**Accuracy**
KNN	0.97	0.87	0.91	0.98
DTREE-CART	0.96	0.90	0.93	0.98
NB	0.73	0.70	0.71	0.91
MLP	0.96	0.89	0.92	0.98
SVM	**0.97**	**0.90**	**0.93**	**0.98**
RFOREST	0.97	0.88	0.92	0.98

The Table III shows the implementation results for all the machine learning algorithms with precision and the F1 macro score. The best performing algorithm obtained an accuracy of 0.98 and a score of 0.93 for the F1 macro. Fig. 3 shows that the SVM algorithm is better than the others. We can therefore conclude that the proposed model obtains results that show its good performance in detecting vulnerabilities. It is effective and can therefore be used to improve the protection of data passing through Android applications.

V. Conclusion

Although identifying malware is a difficult and tedious task, it remains an imperative when it comes to protecting data. Our solution, based on application execution traces and machine learning techniques, meets this challenge. On the one hand, the results obtained enable us to confirm the relevance of execution traces in analysing unexpected and unhealthy application behaviour. On the other hand, these experimental results also show that static and behavioural characteristics are more effective and efficient for detecting malware. The use of behavioural features can enable the detection of malware that escapes the control of solutions based on signatures or static approaches. This compensates for the shortcomings of static feature-based approaches.

In this way, the combination of behavioural and static features improves the level of detection of software vulnerabilities. Our model will make it possible to reduce new threats targeting Android applications. Unfortunately, there are a few limitations to our study, the future objective of which is to generalise this solution to all emerging applications and technologies. Also, to increase the number of applications and consequently the number of traces. The second objective is to have a real environment and not an emulated environment for a better user experience of this solution. We will continue to improve this approach in order to have a vulnerability detection system that is accessible to users.

References

[1] E. Amer and S. El-Sappagh, *Robust deep learning early alarm prediction model based on the behavioural smell for android malware*,Computers & Security, Volume 116, 2022, 102670.

[2] D. T. Dehkordy, and A. Rasoolzadegan *A new machine learning-based method for android malware detection on imbalanced dataset*, Outils et applications multimédias le volume 80 , pages 24533–24554, 2021.

[3] K. D. T. Nguyen, T. M. Tuan, S. H. Le, A. P. Viet, M. Ogawa and N L Minh, *Comparison of Three Deep Learning-based Approaches for IoT Malware Detection*, 10th International Conference on Knowledge and Systems Engineering, 2018.

[4] S. Arshad , A. Khan , M. A. Shah and M. Ahmed, *Android Malware Detection & Protection: A Survey*, (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 7, No. 2, 2016.

[5] G. Lin , S. Wen, Q-L. Han , J. Zhang, And Y. Xiang ”Software Vulnerability Detection Using Deep Neural Networks: A Survey”. DOI: htpps://10.1109/JPROC.2020.2993293, PROCEEDINGS OF THE IEEE, May 2020.

[6] G. Koala, D. Bassolé, A. Zerbo/Sabané, T. F. Bissyandé and O. Sié, ”Analysis of the Impact of Permissions on the Vulnerability of Mobile Applications”. International Conference on e-Infrastructure and e-Services for Developing Countries. AFRICOMM 2019: pp 3–14, dec, 2019.

[7] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Népal and Y. Xiang, *A Survey of Android Malware Detection with Deep Neural Models*, ACM Computing Surveys Volume 53 Numéro 6 Article : 126 pp 1–36, 06 décembre 2020.

[8] statique : W. Wang, Z. Gao, M. Zhao, Y. Li, J. Liu and X. Zhang, *DroidEnsemble: Detecting Android Malicious Applications With Ensemble of String and Structural Static Features*, in IEEE Access, vol. 6, pp. 31798-31807, 2018.

[9] T. Chen, Q. Mao, Y. Yang, M. Lv and J. Zhu, *TinyDroid: A Lightweight and Efficient Model for Android Malware Detection and Classification*, Mobile Information Systems, vol. 2018, Article ID 4157156, 9 pages, 2018.

[10] A. A. Al-Hashmi, F. A. Ghaleb, A. Al-Marghilani, A. E. Yahya, S. A. Ebad, M. S. MS and A. A. Darem, *Deep-Ensemble and Multifaceted Behavioral Malware Variant Detection Model*, in IEEE Access, vol. 10, pp. 42762-42777, 2022.

[11] A. Razagallah, R. Khoury, J-B. Poulet, *TwinDroid: a dataset of Android app system call traces and trace generation pipeline*, MSR ’22: Proceedings of the 19th International Conference on Mining Software Repositories, Pages 591–595, May 2022.

[12] P. L. Cueva, A. Bertaux, A. Termier, J. F. Méhaut, and M. Santana, *Debugging embedded multimedia application traces through periodic*

pattern mining, EMSOFT '12: Proceedings of the tenth ACM international conference on Embedded software, 2012 Pages 13–22.

[13] A. Lebis ,*Capitaliser les processus d'analyse de traces d'apprentissage : modélisation ontologique et assistance à la réutilisation*, Thèse,Sorbonne Université, 2020.

[14] F. Hojaji, T. Mayerhofer, B. Zamani, A. Hamou-Lhadj, and E. Bousse ,*Model execution tracing: a systematic mapping study*, Springer-Verlag GmbH Germany, part of Springer Nature, 2019.

[15] T. Galli ,F. Chiclana, and F. Siewe, *Quality Properties of Execution Tracing, an Empirical Study*, Appl. Syst. Innov. 2021, 4, 20.

[16] G. Koala, D. Bassolé, T. Tiendrebeogo and O. Sié, *Study of an Approach Based on the Analysis of Computer Program Execution Traces for the Detection of Vulnerabilities*. In: Mambo, A.D., Gueye, A., Bassioni, G. (eds) Innovations and Interdisciplinary Solutions for Underserved Areas. InterSol 2022.

[17] S. M. Ghaffarian and H. R. Shahriari, *Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey*, ACM Comput. Surv., vol. 50, no. 4, pp. 1–36, Nov. 2017.

[18] D. Bassolé, Y. Traoré, G. Koala, F. Tchakounté, and O. Sié, *Detection of Vulnerabilities Related to Permissions Requests for Android Apps Using Machine Learning Techniques*. In: , et al. Proceedings of the 12th International Conference on Soft Computing and Pattern Recognition (SoCPaR 2020). Advances in Intelligent Systems and Computing, vol 1383. Springer, Cham, dec, 2020.

[19] M. Odusami, O. Abayomi-Alli, S. Misra, O. Shobayo, R. Damasevicius and R. Maskeliunas, *Android Malware Detection: A Survey*, In: H. Florez, C. Diaz, J. Chavarriaga, (eds) Applied Informatics. ICAI 2018. Communications in Computer and Information Science, vol 942. Springer, Cham, 2018.

[20] A. Razgallah and R. Khoury, *Behavioral classification of Android applications using system calls*, 2021 28th Asia-Pacific Software Engineering Conference (APSEC), Taipei, Taiwan, 2021, pp. 43-52, 2021.

[21] N. A. Hassan, and R. Hijazi, *Digital Privacy and Security Using Windows*, Berkeley: CA Apress, 2017.

[22] D. Zhou, Z. Yan, Y. Fu, and Z. Yao, *A survey on network data collection*, 2018. Journal of Network and Computer Applications 116, pp 9-23, 2018.

[23] J. Lazar, J.H. Feng and H. Hochheiser, *Chapter 12 – Automated data collection methods*,2017. Research Methods in Human Computer Interaction, 2nd edition, pp 329-368, 2017.

[24] F. Gruber, *Performance Debugging Toolbox for Binaries: Sensitivity Analysis and Dependence Profiling* , pp 3-10, 2020.

[25] A. Belkhiri, *Analyse de performances des réseaux programmables, à partir d'une trace d'exécution*, 2021.

[26] H. Venturi, *Le débogage de code optimisé dans le contexte des systèmes embarqués*, pp 13-40.

[27] O. Iegorov, *Data Mining Approach to Temporal Debugging of Embedded Streaming Applications*, pp 89-95, 2018.

[28] , Y. J. Bationo, *Analyse de performance des plateformes infonuagiques*. École Polytechnique de Montréal, pp 19-28, 2016.

[29] F. Reumont-Locke, *Méthodes efficaces de parallélisation de l'analyse de traces noyau*, 2015.

[30] A. Ravanello, *Modeling end user performance perspective for cloud computing systems using data center logs from big data technology*. Thesis, 2017.

[31] C. D. Sestili, W. S. Snavely and N. M. VanHoudnos, *Towards security defect prediction with AI*, arXiv:1808.09897, 2018.

[32] A. Fernández, S. García, M. Galar, R. C. Prati, B. Krawczyk and F. Herrera, *Imbalanced classification for big data*. In: Learning from imbalanced data sets, pp 327–349, Springer, 2018.