View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Archivio istituzionale della ricerca - Università di Trieste

VizMal: A Visualization Tool for Analyzing the Behavior of Android Malware

Alessandro Bacci[1], Fabio Martinelli[2], Eric Medvet[1] and Francesco Mercaldo[2]
[1]*Dipartimento di Ingegneria e Architettura, Università degli Studi di Trieste, Trieste, Italy*
[2]*Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Pisa, Italy*
{abacci, emedvet}@units.it, {fabio.martinelli, francesco.mercaldo}@iit.cnr.it

Keywords: Malware analysis, Android, Machine Learning, Multiple Instance Learning

Abstract: Malware signature extraction is currently a manual and a time-consuming process. As a matter of fact, security analysts have to manually inspect samples under analysis in order to find the malicious behavior. From research side, current literature is lacking of methods focused on the malicious behavior localization: designed approaches basically mark an entire application as malware or non-malware (i.e., take a binary decision) without knowledge about the malicious behavior localization inside the analysed sample. In this paper, with the twofold aim of assisting the malware analyst in the inspection process and of pushing the research community in malicious behavior localization, we propose VizMal, a tool for visualizing the dynamic trace of an Android application which highlights the portions of the application which look potentially malicious. VizMal performs a detailed analysis of the application activities showing for each second of the execution whether the behavior exhibited is legitimate or malicious. The analyst may hence visualize at a glance when at to which degree an application execution looks malicious.

1 Introduction

In recent years, mobile phones have become among the favorite devices for running programs, browsing websites, and communication. Thanks to their ever increasing capabilities, these devices often represent the preferred gateways for accessing sensitive assets, like private data, files and applications, and to connectivity services. On the other hand, this trend also stimulated malware writers to target the mobile platform.

Effective mobile malware detection methods and tools are needed in order to protect privacy and to enable secure usage of mobile phones and tablets. Many detection methods rely on static analysis, i.e., they base on the investigation of static features that are observed before running the application (e.g., occurrences of opcodes in disassembled code (Medvet and Mercaldo, 2016), API usage in code (Aafer et al., 2013)): these techniques are effective under controlled conditions, but may be often easily circumvented by means of malware obfuscation techniques (Egele et al., 2012; Moser et al., 2007). In order to evade current malware detection techniques and in the context of an ongoing adversarial game, malware writers in facts implement increasingly sophisticated techniques (Zhou and Jiang, 2012; Canfora et al., 2014). During its propagation, malware code changes its structure (Canfora et al., 2015d), through a set of transformations, in order to elude signature-based detection strategies (Maiorca et al., 2017; Dalla Preda and Maggi, 2017; Cimitile et al., 2017). Indeed, polymorphism and metamorphism are rapidly spreading among malware targeting mobile applications (Rastogi et al., 2014).

Beyond limitations related to code obfuscation, static analysis detection faces also some limitations peculiar to the Android platform. Indeed, one of the main problems is given by how Android manage applications permissions for observing the file system. Common antimalware technologies derived from the desktop platform exploit the possibility of monitoring the file system operations: this way, it is possible to check whether some applications assume a suspicious behavior; for example, if an application starts to download malicious code, it will be detected immediately by the antimalware responsible for scanning the disk drive. On the other hand, Android does not allow for an application to monitor the file system: every application can only access its own disk space. Resource sharing is allowed only if expressly provided by the developer of the application. Therefore Android antimalware cannot monitor the file system: this allows applications to download updates and run new code without

any control by the operating system. This behavior will not be detected by antimalware software in any way; as a matter of fact, a series of attacks are based on this principle (the so-called "update-attack" (Zhou and Jiang, 2012)).

Despite its limitations, a rather trivial static detection as signature-based detection is often the most common technique adopted by commercial antimalware for mobile platforms. Beyond its low effectiveness, this method of detection is costly, as the process for obtaining and classifying a malware signature is laborious and time-consuming.

In order address the limitations of static analysis, and hence to be able to cope with a variety of malware that exists in the wild, detection based on dynamic analysis is needed: in this case the detection is based on the investigation of dynamic features, i.e., features which can be observed while the application is running (e.g., device resource consumption (Canfora et al., 2016), frequencies of system calls (Martinelli et al., 2017a)).

Most of the currently proposed dynamic detection methods provide the ability to classify applications as malicious or benign as a whole, i.e., without providing any insight on which parts of the application executions are actually malicious. Moreover, trying to manually inspect the features on the base of which those tools take their decision is hard, mainly because of the size of the data. For example, in the many approaches of dynamic detection based on Machine Learning techniques applied to execution traces, the raw data consists of thousands of numbers which are very hard even to be visualized, leaving aside the possibility of being comprehended—this is, indeed, a common issue and interesting line of research in the field of visualization of big data (Fiaz et al., 2016). It follows that an analyst who aims at gaining a better understanding of malware behavior obtains little help from these methods.

Starting from these considerations, in this paper we propose VizMal, a tool for assisting the malware analyst and helping him comprehend the nature of the malware application under analysis. VizMal operates on an execution trace of an Android application and visualizes it as a sequence of colored boxes, one box for each second of duration of the execution. Each box delivers two kinds of information to the analyst: first, an indication, by means of box fill color, of the degree to which the behavior performed by the application during the corresponding second looks malicious; second, an indication, by means of the shape of the box, of how active was the application in the corresponding second.

VizMal may be a valuable tool for Android malware analysts, researchers, and practitioners from many tasks. For example, it can be used to easily spot similarities of behavior between the analyzed application and a well known samples. Or, it can be used together with a tool for executing applications in a controlled environment to find and understand the relation between injected events (e.g., an incoming SMS) and observed behavior with the aim of, e.g., looking for the payload activation mechanism. Or, finally, it can be used to debug a malware detection method by performing a fine-grained analysis of misclassified applications.

Internally, VizMal analyzes the execution trace using a *Multiple Instance Learning* (MIL) framework. Multiple Instance Learning (Carbonneau et al., 2016; Zhou, 2004) is a form of weakly-supervised learning in which instances in the learning set are grouped and the label is associated with a group, instead of with a single instance. The MIL framework fits our scenario because labeling execution traces of Android applications at the granularity of one or few seconds (subtraces) is very costly: hence data for training a classifier capable of classifying subtraces would be hard to collect and even harder to keep updated. MIL addresses this issue by considering each subtrace as an instance and an entire trace as a group of instances for which a label exist and is easily obtainable, e.g., by running several known Android malware application for a long enough time. To the best of our knowledge, this is the first use of MIL in the context of Android malware classification.

The remaining of the paper is organized as follows. In Section 2 we briefly review the state-of-the-art on malware Android detection; in Section 3 we describe how the tool (VizMal) works; in Section 4 we discuss the results of the experimental validation of VizMal, including a comparison of some alternatives for the main components of VizMal; finally, in Section 5, we draw some conclusions and propose future lines of research.

2 Related work

In this section we review the current literature related to the Android malware detection topic with particular regards to methods focused on, or possibly able to perform, the localization of the malicious behaviors (overcoming the classic malware detector binary output, i.e., malware/non-malware).

Amandroid (Wei et al., 2014) performs an inter component communication (ICC) analysis to detect leaks. Amandroid needs to build an Inter-component Data Flow Graph and an Data Dependence Graph to perform ICC analysis. It is basically a general frame-

work to enable analysts to build a customized analysis on Android apps.

FlowDroid (Arzt et al., 2014) adequately models Android-specific challenges like the application lifecycle or callback methods. It helps reduce missed leaks or false positives: the proposed on-demand algorithms allow FlowDroid to maintain efficiency despite its strong context and object sensitivity.

Epicc (Octeau et al., 2013) identifies a specification for every ICC source and sink. This includes the location of the ICC entry point or exit point, the ICC Intent action, data type and category, as well as the ICC Intent key/value types and the target component name.

Mercaldo et al. (2016); Canfora et al. (2015f,e) evaluate the effectiveness of the occurrences of a subset of opcodes (i.e., `move`, `if`, `jump`, `switch`, and `goto`) in order to discriminate mobile malware applications from non-malware ones. They apply six classification algorithms (J48, LADTree, NBTree, RandomForest, RandomTree, and RepTree), obtaining a precision equal to 0.949 in malware identification.

These methods consider static analysis in order to identify the threats: as discussed into the introduction, using static analysis it is possible to identify malicious payloads without infect the device under analysis, but these techniques exhibit a strong weakness with respect to the code obfuscation techniques currently employed by malware writers (Canfora et al., 2015b).

The approach presented by Ferrante et al. (2016) exploits supervised and unsupervised classification in order to identify the moment in which an application exhibits a malware behavior. Despite the general idea and aim are similar to those of the present work, the cited paper lacks the visualization component and hence can hardly be used directly by the analyst.

The Andromaly framework (Shabtai et al., 2012) is based on a Host-based Malware Detection System able to continuously monitor features (in terms of CPU consumption, number of sent packets through the Wi-Fi, number of running processes and battery level) and events obtained from the mobile device and consider machine learning to classify the collected data as normal (benign) or abnormal (malicious). The proposed solution is evaluated on four applications developed by authors.

BRIDEMAID (Martinelli et al., 2017b) is a tool able to combine static and dynamic analysis in order to detect of Android Malware. The analysis is based on multi-level monitoring of device, app and user behavior with the aim to detect and prevent at runtime malicious behaviors.

AndroDialysis (Feizollah et al., 2017) considers Android Intents (explicit and implicit) as a distinguishing feature for malware identification. The results show that the use of Android Intent achieves a better detection ratio if compared with the permission analysis.

TaintDroid (Enck et al., 2014) is an extension to the Android operating system that tracks the flow of privacy sensitive data through third-party applications. TaintDroid assumes that downloaded, third-party applications are malware, and monitors in realtime how these applications access and manipulate users' personal data.

Researchers in (Tam et al., 2015) consider system call extraction with the aim to generates behavioral profiles of Android applications. The developed framework, i.e. CopperDroid, is able to automatically reconstruct system call semantics, including IPC, RPC, and Android objects.

Lindofer et al. (Lindorfer et al., 2015) discuss the MARVIN tool, an analysis tool able to assess the maliciousness of Android applications. MARVIN considers machine learning techniques to classify Android mobile applications using an extended feature set extracted from static and dynamic analysis of a set of known malicious and benign applications.

These techniques consider dynamic analysis in order to label Android samples as malware or trusted: in detail the methods presented in (Tam et al., 2015) and Ferrante et al. (2016) exploit the syscall traces as discriminant between malware and legitimate samples as VizMal. The main difference between VizMal and the methods designed in (Tam et al., 2015) and Ferrante et al. (2016) is the visualization component that make our method useful to malware analyst in order to automatically and quickly find the malicious behaviour.

3 The proposed tool: VizMal

We consider the visual evaluation of malware Android apps for fast and precise individuation of malicious behaviors during the execution. To reach this goal, we built a visualization tool that takes as input an execution trace t of an app and outputs an image consisting of a sequence of colored boxes.

Each box corresponds to a period lasting T seconds of the execution trace t; the value of T is a parameter of the tool: in this work we focused on the case of $T = 1\,\mathrm{s}$ which is a good trade-off between informativeness and easiness of comprehension. The color of the box is related to the degree to which the behavior performed by the application during the corresponding second looks malicious. The shape of the box (in particular, its height) is related to how active was the application

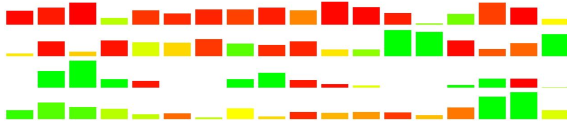


Figure 1: Examples of the images obtained by VizMal on 4 execution traces of malware apps: only the first 18 s are here depicted.

in the corresponding second.

VizMal is composed by two components: an *image builder*, which builds the image, and a *trace classifier*, which processes the trace t and decorates it with its maliciousness and activity levels. As briefly introduced in Section 1, the trace classifier is based on MIL: before being able to process execution traces, it has to be trained on a set of labeled execution traces (one label in {malware, non-malware} for each trace). In the following sections, we describe the two components.

3.1 Image builder

The input of the image builder is a sequence $L = \{(m_1, a_1), (m_2, a_2), \dots\}$ of pairs of values. The i-th pair refers to the subtrace of the trace t starting at $(i-1)T$ second and ending at iT second: $m_i \in [0,1]$ is the *maliciousness level* of that subtrace (0 means no maliciousness) and $a_i \in [0,1]$ represents the *activity level* of that portion (0 means no activity).

The image is composed of a vertical sequence (i.e., a row) of boxes, one for each element in the input sequence P. Boxes have the same width w and, for the sake of clarity, are separated by a small empty gap. The height of the i-th box is $a_i w$, where w is the box width and a_i is the activity level of the corresponding subtrace. The fill color of the box is solid and determined basing on m_i: for $m_i = 0$, it is green, for $m_i = 1$ it is red, and for intermediate values, it is given by the point on a line connecting green and red in the HSL color space whose distance from green is m_i assuming than the length of the line is exactly 1.

Figure 1 shows 4 images obtained with VizMal applied to execution traces of malware apps: different maliciousness and activity levels can be seen in the color and height of the boxes.

3.2 Trace classifier

The trace classifiers operates in two phase. First, it must be trained in a *learning phase* which takes as input a set of labeled execution traces; then, it can be used in the *classification phase* for actually generating a sequence L of maliciousness and activity levels out of an execution trace t. In both phases, each execution trace is preprocessed in order to extract some features: in this work, we were inspired by the approach proposed by Canfora et al. (2015c) where features are frequences of n-grams of the system calls occurred in the trace. We remark, however, that any other approach able to provide a sequence L of maliciousness and activity levels out of an execution trace t could also apply.

In detail, the preprocessing of a trace t is as follows. The trace classifier (i) splits the trace t in a sequence $\{t_1, t_2, \dots\}$ of subtraces, with each subtrace lasting exactly T seconds; (ii) considers subsequences (n-grams) of at most N consecutive system calls (discarding the arguments), where N is a parameter of the trace classifier; (iii) counts the number $o(t_i, g)$ of each n-gram g in each subtrace t_i.

In the learning phase, the trace classifiers trains a MIL-based binary classifier using substraces as instances, the label (non-malware or malware) of their enclosing trace as group label, and the counts $o(t_i, g)$ of n-grams as features.

In the classification phase, the trace classifiers first preprocesses the input trace t obtaining the subtraces and the corresponding feature values. Then, it classifies each subtrace using the trained MIL-based classifier and obtaining a label with a confidence value. Finally, it sets the value of the maliciousness level m_i for each subtrace t_i according to the assigned label and corresponding confidence value; and it sets the value of the activity level to $a_i = \frac{|t_i|}{\max_i |t_i|}$, i.e., to the ratio between the number of system calls in the subtrace and the maximum number of system calls in a subtrace of t.

According to the findings of Canfora et al. (2015c), in this work we set $N = 1$: in other words, we considered the absolute frequencies of unigrams—we remark, however, that more sophisticated features could be used. Concerning the MIL-based classifier, we used *miSVM* (Andrews et al., 2003) with a linear kernel and the parameter C parameter set to 1.

4 Validation

We performed a set of experiments to validate our proposal, i.e., to verify that VizMal may actually help the analyst in better understanding malware (and non malware) apps behavior.

To this end, we considered a dataset of 200 Android apps (a subset of those used in (Canfora et al., 2015a)), including 100 non-malware apps automatically downloaded from the Google Play Store and 100 malware apps from the Drebin dataset (Arp et al., 2014). For each app in the dataset, we obtained 3 execution traces by executing the app for (at most) 60 s

on a real device with the same procedure followed by Canfora et al. (2015c).

In order to simulate the usage of VizMal to analyze new, unseen malware, we divided the dataset in a set of 180 and a set of 20 apps. We first trained the tool on the 180×3 traces corresponding to the former set and then applied it to the 20×3 traces of the latter obtaining several images. We repeated the procedure several times by varying the dataset division and obtained consistent results: we here report a subset of the images obtained in one repetition.

Figures 2 and 3 show the images obtained by VizMal applied to the 3 traces of 3 malware and 3 non-malware app, respectively. Several interesting observation may be made.

Concerning the malware traces in Figure 2, it can be seen that the images present several red boxes representing seconds classified as malware with a high confidence, but also some green ones, which indicate seconds with no malware behaviors recognized. Furthermore, some yellow and orange boxes show uncertain seconds of execution: these seconds are classified as non-malware (in yellow) and as malware (in orange) with a lower confidence. The height of the boxes shows a variable activity during the execution, going from seconds with a very high number of system calls to seconds with almost no activity. Finally, the different number of boxes in the images in Figure 2 indicates that in many cases malware apps stopped the execution before the 60 s time limit: we verified that this finding is due to the machinery used to collect the execution traces, in which random user interaction events were simulated by means of an ad hoc tool (Canfora et al., 2015c).

It can be seen that, with the proposed tool, it is immediately observable when an app exhibits a general malicious behavior (e.g., last 2 on 3 traces for the second malware app in Figure 2b) or when it behaves "normally" (e.g., second trace of third malware app in Figure 2c) or "borderline" (e.g., first trace of first malware app in Figure 2a). Moreover the exact moments during which the malware behavior occurs and its intensity can be easily identified. These information allows for a detailed analysis of the app.

Similar considerations can be made for the images obtained for non-malware apps of 3. A row of green boxes indicates that the app behavior was normal during the entire period of execution of 60 s. It can also be seen, from second row of Figure 3b, that it may happen that an non-malware app behavior looks suspicious from the point of view of the sequences of system calls. This can be an opportunity, for the analyst, to gain more insights in the execution trace or on the classification machinery.

We remark that VizMal applies to a single execution trace: hence, issues related to the representativeness of such a single trace of the behavior of an app in general (e.g., code coverage) are orthogonal to the goal of VizMal. However, since VizMal allows the analyst to quickly analyze a single execution trace, it may also enable a faster analysis of several traces collected from the same app, perhaps in order to maximize the generality of findings, e.g., w.r.t. code coverage.

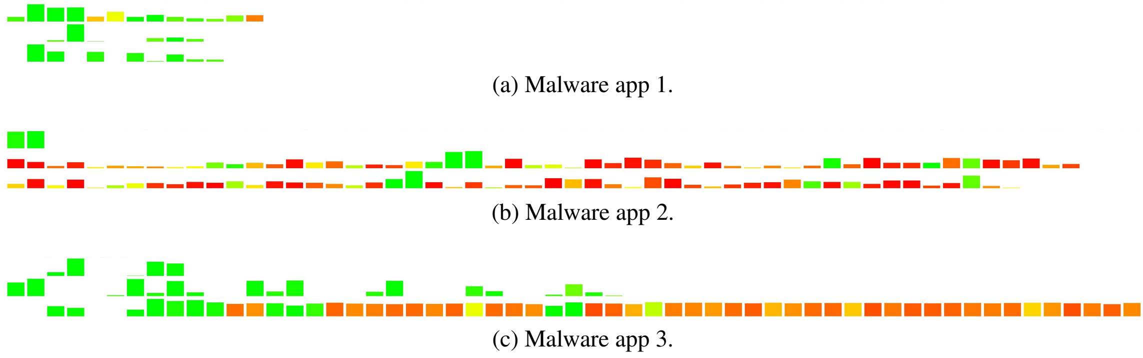
4.1 Alternative image builders

Before converging on the final proposed visualization (visible in Figure 1), we explored many design options for the image builder, progressively adding elements to enrich the information displayed while keeping the image easily readable. We here present the most significant variants.

We started with the simplest configuration, i.e., where boxes delivered a binary indication (green or red) for the maliciousness level and no indication for the activity level. The result is shown in Figure 4 for 3 execution traces of malware apps (top) and one trace of a non-malware app (bottom). The first three apps can be correctly recognized as malware. The last one instead might look like a malware app (even though the malware activity is much shorter compared with the others apps), but it is actually a non-malware app. This example shows that using only two colors for describing all the information makes the analysis very limited.

To display more information, we decided to include in the visualization also an indication of the activity level, as the number of system calls executed during each subtrace. An example of this visualization is shown in Figure 5, for the same traces of Figure 4. With this visualization, it is easy to see that in the non-malware app the malware activity is in fact lower, i.e., fewer system calls were executed during the seconds classified as malware with respect to the actual malware apps. This might suggest an erroneous evaluation from the underlying classifier, but it the claim would be very weak.

We tried another approach and encoded the confidence of the classification using the fill color of the boxes, i.e., basing on m_i as described in Section 3.1. Figure 6 shows the result for the same traces of Figure 4. The fill color shows that the confidences of maliciousness are lower in the non-malware app, for which there are no red boxes. Malware apps instead contain many boxes with a high confidence of maliciousness. This consideration can lead to the hypothesis of a false positive of the trace classifier or, from another point of view, of borderline behavior.

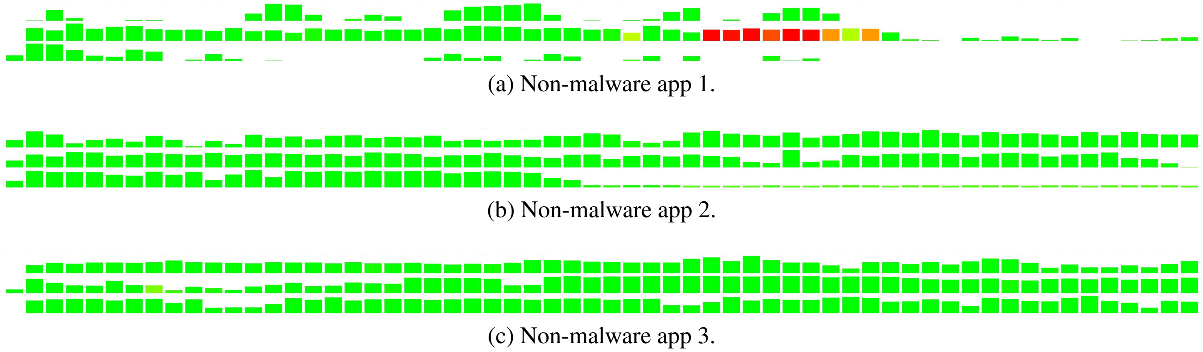


(a) Malware app 1.

(b) Malware app 2.

(c) Malware app 3.

Figure 2: Images obtained from the traces of 3 malware apps.



(a) Non-malware app 1.

(b) Non-malware app 2.

(c) Non-malware app 3.

Figure 3: Images obtained from the traces of 3 non-malware apps.

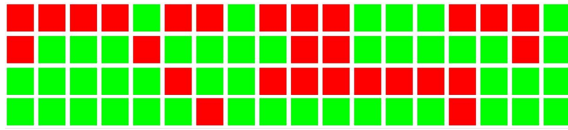


Figure 4: Examples of the images obtained by the VizMal variant with binary maliciousness level and no activity level for 3 malware traces and one non-malware trace.

Figure 5: Examples of the images obtained by the VizMal variant with binary maliciousness level and activity level related to number of system calls for the same execution traces of Figure 4.

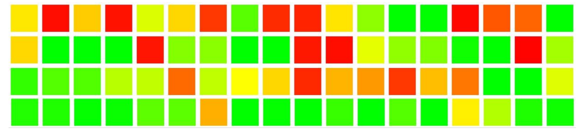


Figure 6: Examples of the images obtained by the VizMal variant with continuous maliciousness level and no activity level for the same execution traces of Figure 4.

Figure 7: Examples of the images obtained by the VizMal variant with continuous maliciousness level and activity level related to number of system calls for the same execution traces of Figure 4

Eventually, we converged to the proposed solution for the image builder component of VizMal. The fill color is related to the maliciousness level m_i in a continuous way and the box shape is related to the activity level a_i. Figure 7 shows the image obtained in this (final) variant for the same traces of Figure 4. The non-malware app representation is different enough from the malware apps to indicate a probable wrong classification and the necessity to perform a deeper analysis to clarify the nature of the app.

4.2 Alternative trace classifiers

In order to explore different options for the MIL-based classifier on which the trace classifier is built, we considered 4 other algorithms able to work in the considered scenario. We remark that the reason for which VizMal internally bases on a MIL framework is that because obtaining a dataset of traces annotated with a malware/non-malware label with the granularity of one or few seconds is costly. Instead, using MIL, Viz-

Mal can train on traces obtained by malware and non-malware apps without particular constraints, under the assumption that the malware behavior eventually occurs if the execution is long and varied enough.

We experimented with sMIL (Bunescu and Mooney, 2007), miSVM (Andrews et al., 2003), and an ad hoc variant of "single instance" SVM (SIL-SVM) which we modified in order to act as a MIL framework. For the latter, we built a learning set in which we applied a malware label to each subtrace of a trace corresponding to a malware app and a non-malware label to each subtrace of a trace corresponding to a non-malware app. For sMIL and miSVM, instead, a label is associated with an entire trace, with the semantic that a malware label means that at least one subtrace is malware, whereas a non-malware label means that all the subtraces are non-malware.

In order to assess the 3 variants, we performed the following procedure:

1. we divided the dataset of $3 \times 100 + 3 \times 100$ execution traces (see Section 4) in a balanced learning set composed of 90% of the traces and a testing set composed of the remaining traces;
2. we trained the three classifiers on the learning set;
3. we applied the trained classifiers on the traces in the testing set.

We repeated the above procedure 5 time by varying the learning and testing set compositions and measured the performance of the classifiers as False Positive Rate (FPR), i.e., ratio between the number of subtraces of non-malware apps classified as malware and the number (30×60) of all the non-malware subtraces, and False Negative Rate (FNR), i.e., ratio between the number of subtraces of malware apps classified as non-malware and the number of all (30×60) the malware subtraces. Since all the considered MIL variants base on SVM, for a fair comparison we used the linear kernel and $C = 1$ for all.

Table 1 presents the results, averaged across the 5 repetitions. We remark that (a) our experimentation was not aimed at performing a comparison among MIL frameworks—the interested reader may refer to (Ray and Craven, 2005)—and (b) the shown figures should not be intended as representative of the accuracy of malware detection for the considered approaches. In facts, while it is fair to consider a false positive (i.e., a subtrace of a non-malware app classified as a malware) as an error, the same cannot be done for a false negative (i.e., a subtrace of a malware app classified as a non-malware): it may actually occur, possibly with high probability, that even a malware app does not exhibit a malicious behavior for some seconds during its execution.

Table 1: FPR and FNR (in percentage) for three considered variants of MIL classifiers.

Classifier	FPR	FNR
SIL-SVM	75.80	11.28
SIL-SVM (near EER)	40.24	38.48
miSVM	26.44	42.44
sMIL	9.80	69.28
sMIL (near EER)	30.84	36.36

It can be seen from Table 1 that miSVM (the variant which we used in VizMal) outperforms both SIL-SVM and sMIL. The values of FPR and FNR for the latters suggest that their output is biased towards the malware (SIL) and non-malware (sMIL) labels. To mitigate this effect, we tuned the threshold of the to classifiers in order to obtain their effectiveness indexes in a working point close to the Equal Error Rate (EER), also reported in Table 1. However, it can be seen that miSVM still appears as the most effective variant.

5 Concluding remarks

In this paper we introduced VizMal, a tool that presents in a graphical way the results of a dynamic malware analysis of Android applications. VizMal takes an execution trace of an Android application and shows a row of colored boxes, one box for each second of duration of the execution: the color of the box represents the maliciousness level of the app during the corresponding second, whereas the box shape represents the app activity level during the corresponding second. VizMal may be a valuable tool in the Android malware analysts' and researchers' toolboxes, allowing them to better comprehend the nature of malware application and debug other, maybe more sophisticated, detection systems. Along this line, we intend to explore the use of VizMal together with tools for controlled execution of Android apps in order to investigate the possibility of relating the injected user and system events to the maliciousness and activity levels measured by VizMal, possibly resulting in an interactive tool for malware analysis.

Acknowledgments

This work has been partially supported by H2020 EU-funded projects NeCS and C3ISP and EIT-Digital Project HII.

REFERENCES

Aafer, Y., Du, W., and Yin, H. (2013). Droidapiminer: Mining api-level features for robust malware detection in android. In *International Conference on Security and Privacy in Communication Systems*, pages 86–103. Springer.

Andrews, S., Tsochantaridis, I., and Hofmann, T. (2003). Support vector machines for multiple-instance learning. In Becker, S., Thrun, S., and Obermayer, K., editors, *Advances in Neural Information Processing Systems 15*, pages 577–584. MIT Press.

Arp, D., Spreitzenbarth, M., Huebner, M., Gascon, H., and Rieck, K. (2014). Drebin: Efficient and explainable detection of android malware in your pocket. In *Proceedings of 21th Annual Network and Distributed System Security Symposium (NDSS)*.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269.

Bunescu, R. C. and Mooney, R. J. (2007). Multiple instance learning for sparse positive bags. In *Proceedings of the 24th Annual International Conference on Machine Learning (ICML-2007)*, Corvallis, OR.

Canfora, G., De Lorenzo, A., Medvet, E., Mercaldo, F., and Visaggio, C. A. (2015a). Effectiveness of opcode ngrams for detection of multi family android malware. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 333–340. IEEE.

Canfora, G., Di Sorbo, A., Mercaldo, F., and Visaggio, C. A. (2015b). Obfuscation techniques against signature-based detection: a case study. In *2015 Mobile Systems Technologies Workshop (MST)*, pages 21–26. IEEE.

Canfora, G., Medvet, E., Mercaldo, F., and Visaggio, C. A. (2015c). Detecting android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 13–20. ACM.

Canfora, G., Medvet, E., Mercaldo, F., and Visaggio, C. A. (2016). Acquiring and analyzing app metrics for effective mobile malware detection. In *Proceedings of the 2016 ACM International Workshop on International Workshop on Security and Privacy Analytics*. ACM.

Canfora, G., Mercaldo, F., Moriano, G., and Visaggio, C. A. (2015d). Composition-malware: building android malware at run time. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 318–326. IEEE.

Canfora, G., Mercaldo, F., and Visaggio, C. A. (2014). Malicious javascript detection by features extraction. *e-Informatica Software Engineering Journal*, 8(1).

Canfora, G., Mercaldo, F., and Visaggio, C. A. (2015e). Evaluating op-code frequency histograms in malware and third-party mobile applications. In *E-Business and Telecommunications*, pages 201–222. Springer.

Canfora, G., Mercaldo, F., and Visaggio, C. A. (2015f). Mobile malware detection using op-code frequency histograms. In *Proceedings of International Conference on Security and Cryptography (SECRYPT)*.

Carbonneau, M.-A., Cheplygina, V., Granger, E., and Gagnon, G. (2016). Multiple instance learning: A survey of problem characteristics and applications. *arXiv preprint arXiv:1612.03365*.

Cimitile, A., Martinelli, F., Mercaldo, F., Nardone, V., and Santone, A. (2017). Formal methods meet mobile code obfuscation identification of code reordering technique. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2017 IEEE 26th International Conference on*, pages 263–268. IEEE.

Dalla Preda, M. and Maggi, F. (2017). Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. *Journal of Computer Virology and Hacking Techniques*, 13(3):209–232.

Egele, M., Scholte, T., Kirda, E., and Kruegel, C. (2012). A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*, 44(2):6.

Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2014). Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5.

Feizollah, A., Anuar, N. B., Salleh, R., Suarez-Tangil, G., and Furnell, S. (2017). Androdialysis: analysis of android intent effectiveness in malware detection. *computers & security*, 65:121–134.

Ferrante, A., Medvet, E., Mercaldo, F., Milosevic, J., and Visaggio, C. A. (2016). Spotting the malicious moment: Characterizing malware behavior using dynamic features. In *Availability, Reliability and Security (ARES), 2016 11th International Conference on*, pages 372–381. IEEE.

Fiaz, A. S., Asha, N., Sumathi, D., and Navaz, A. S. (2016). Data visualization: Enhancing big data more adaptable and valuable. *International Journal of Applied Engineering Research*, 11(4):2801–2804.

Lindorfer, M., Neugschwandtner, M., and Platzer, C. (2015). Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 2, pages 422–433. IEEE.

Maiorca, D., Mercaldo, F., Giacinto, G., Visaggio, C. A., and Martinelli, F. (2017). R-packdroid: Api package-based characterization and detection of mobile ransomware. In *Proceedings of the Symposium on Applied Computing*, pages 1718–1723. ACM.

Martinelli, F., Marulli, F., and Mercaldo, F. (2017a). Evaluating convolutional neural network for effective mobile malware detection. *Procedia Computer Science*, 112:2372–2381.

Martinelli, F., Mercaldo, F., and Saracino, A. (2017b). Bridemaid: An hybrid tool for accurate detection of android malware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 899–901. ACM.

Medvet, E. and Mercaldo, F. (2016). Exploring the usage of topic modeling for android malware static analysis. In *Availability, Reliability and Security (ARES), 2016 11th International Conference on*, pages 609–617. IEEE.

Mercaldo, F., Visaggio, C. A., Canfora, G., and Cimitile, A. (2016). Mobile malware detection in the real world. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 744–746. ACM.

Moser, A., Kruegel, C., and Kirda, E. (2007). Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE.

Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., and Le Traon, Y. (2013). Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. *Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis*.

Rastogi, V., Chen, Y., and Jiang, X. (2014). Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108.

Ray, S. and Craven, M. (2005). Supervised versus multiple instance learning: An empirical comparison. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 697–704, New York, NY, USA. ACM.

Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., and Weiss, Y. (2012). "andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190.

Tam, K., Khan, S. J., Fattori, A., and Cavallaro, L. (2015). Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*.

Wei, F., Roy, S., Ou, X., et al. (2014). Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM.

Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE.

Zhou, Z.-H. (2004). Multi-instance learning: A survey. *Department of Computer Science & Technology, Nanjing University, Tech. Rep*.