# Visualizing the Outcome of Dynamic Analysis of Android Malware with VizMal

Andrea De Lorenzo[a], Fabio Martinelli[b], Eric Medvet[a], Francesco Mercaldo[b,c], Antonella Santone[c]

[a] *Dipartimento di Ingegneria e Architettura, Università degli Studi di Trieste, Trieste, Italy*

[b] *Istituto di Informatica Telematica, Consiglio Nazionale delle Ricerche (CNR), Pisa, Italy*

[c] *Dipartimento di Bioscienze e Territorio, Università degli Studi del Molise, Pesche (IS), Italy*

**Abstract**

Malware detection techniques based on signature extraction require security analysts to manually inspect samples to find evidences of malicious behavior. This time-consuming task received little attention by researchers and practitioners, as most of the effort is on the identification as malware or non-malware of an entire sample. There are no tools for supporting the analyst in identifying when the malicious behavior occurs, given a sample. In this paper we propose VizMal, a tool able to visualize the execution traces of Android applications and to highlight which portions of the traces correspond to a potentially malicious behavior. The aim of VizMal is twofold: assisting the malware analyst during the inspection of an application and pushing the research community to organize and focus its effort on the malicious behavior localization. VizMal is able to discern if the application behavior during each second of execution are legitimate or malicious and to show this information in a simple and understandable way. We validate VizMal experimentally and by means of a user study: the results are promising and confirm that the tool can be useful.

*Keywords:* Malware analysis, Android, Machine Learning, Multiple Instance Learning, LSTM-RNN, Security

## 1. Introduction

Nowadays, most of malware is conveyed by mobile applications, since smartphone and mobile devices in general are the preferred tools to access online resources [53].

Malware writers are encouraged to focus their efforts in continuously developing new and sophisticated techniques for thwarting malware detectors [65]. In this scenario, it is fundamental to find effective mobile malware detection methods and tools [35, 5] in order to address the increasing need of privacy protection and secure usage of mobile platforms.

Many detection methods are based on static analysis, investigating static features observed before running the application (e.g., occurrences of opcodes in disassembled code [47], API usage in code [4]). While these approaches may be very effective, they may also be circumvented with obfuscation techniques easily [11, 27, 50]. Indeed, malicious software writers increasingly implement sophisticated methods [69, 21] for bypassing signature-based detection strategies [42, 25, 23, 16, 57, 19]. Malware code is often able to execute a set of transformations to dynamically change its structure [19], such as in the cases of polymorphic and metamorphic malware [54].

In addition to the limitations in presence of code obfuscation, static analysis detection faces also some weaknesses peculiar to the Android platform. Common antimalware technologies derived from the desktop platform exploit the possibility of monitoring file system operations, to monitor a possible suspicious behavior of an application. For example, if an application starts to download malicious code, this activity will be detected immediately by the antimalware responsible for scanning the disk drive. Nevertheless, Android does *not* allow applications to monitor the file system: each application can only access its own disk space, and resource sharing is allowed only if expressly provided by the software developer. This means that applications can download updates and run new code without any kind of control by the operating system. Malicious behavior of this form ("update-attack" [69, 30, 31]) cannot be detected by antimalware software.

Despite the previously mentioned weaknesses, static detection based on predefined signatures is still the most common approach adopted by commercial antimalware for mobile platforms. Such an approach, moreover, has proven to be quite costly since the procedure for developing a malware signature is laborious and time-consuming.

In order to address the limitations of static analysis, a detection based on dynamic analysis is needed. Typically dynamic analysis is related to process monitoring [10], memory analysis [67, 56] and network traffic [61, 64].

In our work, the detection is based on the investigation of features observed while the application is running (e.g., device resource consumption [18], frequencies of

system calls [43]). Dynamic analysis demonstrated to be more effective in the malware detection task than static analysis, as discussed in [27].

Most of the currently proposed dynamic detection methods are able 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, because of the huge volume of dynamically collected data, trying to inspect those data in order to gain insight into how tools make their decision is very hard. For example, in many approaches of dynamic detection based on Machine Learning, the volume of the raw data makes visualization impractical, leaving aside the possibility of being comprehended—this is, indeed, a common issue and interesting line of research in the field of big data visualization [34]. It follows that an analyst who aims at gaining a better understanding of malware behavior obtains little help from these methods.

In this paper we propose *VizMal*, a tool for supporting the malware analysts in understanding the nature of the malware Android application. VizMal works on an execution trace of an application represented as a sequence of system call invocations and constructs an image consisting of a sequence of colored boxes, one for each interval of execution (corresponding to one second in this work). Each box delivers two kinds of information to the analyst: the *color* represents the *degree of maliciousness* of the execution in that interval, while the *size* represents the *activity level* in terms of number of invoked system calls.

VizMal may be a valuable tool for many tasks, such as to easily spot similarity in the behaviour of the analyzed application and a well known sample. It can also be used together with a tool for executing applications in a controlled environment to find and understand the relationship between injected events (e.g., an incoming SMS) and observed behavior with the aim of, e.g., looking for the payload activation mechanism. 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 *Long Short-Term Memory (LSTM)* neural network which labels each single interval of the execution trace. LSTM networks are a form of RNN (Recurrent Neural Network) that have been proven to be able to learn long term dependencies in the input sequence [36, 68]. The LSTM framework fits our scenario very well because it is able to label sequences of widely varying length effectively. We train an LSTM with long execution traces labelled as malware or non-malware, then we use the trained LSTM for labelling each piece of a long trace. In this way we may associate information

highly useful to an analyst with each single interval of an execution trace. We remark that the purpose of VizMal is not to classify if an application is or is not a malware, but to show in a quick and easily understandable way what is the behavior of such an application.

A very preliminary paper introducing VizMal was presented in [9]. In the present study, we extend the preliminary version in two ways: (i) we introduce and experimentally evaluate a new classification method based on a Long Short-Term Memory recurrent neural network (LSTM-RNN), (ii) we consider more design alternatives for the visualization part and analyze them more in detail, and (iii) we conduct a user study in the form of a questionnaire, filled by 33 users, in order to evaluate if VizMal may serve as a useful tool for malware analysts.

This 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 VizMal experimental validation; in Section 5 we compare some alternatives for the main components of VizMal; in Section 6 we report the results of a user-based study for assessing the user-perceived effectiveness of our tool; finally, in Section 7, we draw some conclusions and propose future lines of research.

## 2. Related work

In this section, we review the current literature related to Android malware detection. We first discuss techniques for malicious payload localization in a mobile environment. Then, we focus on techniques based on the analysis of system calls. It can be seen that the existing literature is mostly focused on the problem of classifying the behavior of an executable as a whole (malware/legitimate). Our work is instead concerned with how to obtain and visualize indications about the localization of malicious behavior in an executable.

### *2.1. Malware detection in mobile environments*

Several methods of static analysis have been proposed for identifying the pattern of interaction among the modules of an application (*inter-component communication, ICC*) and obtaining indications about possible privacy leaks. Amandroid [66] executes an ICC analysis by constructing a Data Flow Graph and a Data Dependence Graph. FlowDroid [8] includes in the ICC analysis both the code and configuration files of the examined app, which may help in reducing both missed leaks and false

positives. Epicc [51] is able to construct a specification for every ICC source and sink, including the location of the ICC entry point or exit point, the ICC Intent action, data type and other information useful for the analysis, e.g., name of the interacting ICCs.

Static analysis has been also proposed for classifying executables (malware vs legitimate).

AndroDialysis [32] proposes a classifier that exploits the rich semantics of *Intents*, i.e., a messaging object of the Android communication system describing the operations that an application intends to perform and that can be extracted from the code. The cited paper shows that the use of Android Intents allows obtaining a better detection ratio that the use of permissions [20].

Methods based on static analysis do not require infecting any device nor do they require any instrumented platform for collecting data during execution in a controlled environment. On the other hand, static methods may be circumvented by code obfuscation techniques rather easily [15, 24, 49, 45, 48].

A classification method based on features collected both statically and dynamically is proposed in [41]. The authors designed a tool able to assess the maliciousness of Android applications. The authors employ a machine learning technique with the aim of classifying Android applications using a feature set gathered from static and dynamic analysis of a set of known malicious and legitimate mobile applications. The BRIDEMAID [46, 29] framework advocates a similar hybrid approach and includes a multi-level monitoring of device, app and user behavior in order to detect malicious behaviors at run time. The Andromaly framework [60] proposes a classification based solely on dynamic features. The framework consists of a form of Host-based Malware Detection System able to monitor features (i.e., CPU consumption, number of packets sent through the network, number of running processes and battery level) and events obtained from the mobile device during execution.

A combination of static and dynamic techniques is used in [12] for analyzing suspicious Android applications. The approach consists, essentially, in executing the application within a sandbox and catching all potentially dangerous events, e.g., file open operations, connections to remote server and so on. The method is assessed on an example application that implements a form of Denial of Service, i.e., a fork bomb which uses *Runtime.Exec()* to start an external binary program and creates sub-processes of itself in an infinite loop.

Several techniques for dynamic analysis have been proposed that consider *power*

*consumption* as the discriminating feature between benign and malicious mobile applications. Authors in [26] hypothesized that there is a strong correlation between mobile devices power consumption pattern and location and studied power consumption data from twenty smartphone users collected over a period of three months. They tested the method on a Nokia 5500 Sport to evaluate real-world mobile malware. Method proposed in reference [39] consists in a power-aware malware-detection framework that monitors and detects unknown energy-depletion threats. Their solution is composed of (1) a power monitor which collects power samples and builds a power consumption history from the collected samples, and (2) a data analyzer which generates a power signature from the constructed history. To generate a power signature, noise-filtering and data-compression are applied, with the aim to reduce the detection overhead. They conducted the experiment using an HP iPAQ running a Windows Mobile OS.

The Jackdaw [52] tool is based on the correlation of control and data-flow information obtained from binaries both statically and dynamically. Basically it is able to find groups of API calls that represent high level actions, exploiting hybrid analysis. Authors evaluated Jackdaw on 2,136 distinct binaries, including both malicious and benign libraries.

TaintDroid [28] is an extension to the Android operating system with the ability to track the flow of privacy sensitive data through third-party applications: basically it assumes that downloaded, third-party applications are malware, and monitors in real time how these applications access and manipulate personal data of users.

Researchers in [59] detect suspicious temporal patterns to decide whether an intrusion is found, using patterns predefined by security experts. They use several features to perform this task, for instance memory, CPU and power consumption, keyboard usage and so on. Their dataset was formed by five Android malicious applications, which were developed by the authors.

The approach presented in [33] 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 this work, the cited paper lacks the visualization component and hence can hardly be used directly by the analyst.

### *2.2. Malware detection exploiting system calls*

CopperDroid demonstrated the possibility of reconstructing the behavior of malicious applications automatically, based on dynamic collection and analysis of

Android system call traces [62]. The resulting framework also allows uncovering malicious behavior through suitable stimulations that can be selected based on reconstructed behavior profiles. The method was validated on a set of 1600 malicious apps, and was able to find the 60% of the malicious apps belonging to one sample (the Genoma Project), and the 73% of the malicious apps included in the second sample (the Contagio dataset).

The proposal in [38] is able to track and examine the flow of sensitive information by monitoring and logging file I/O operations and InterProcess Communication through Intents objects. In detail this method consists of two modules: the first one hooks a function to monitor IPC/RPC events, while the second hooks system calls to trace and log the malicious behavior of a specific application. The technique is implemented using the Loadable Kernel Module programming interfaces and the Kprobes debugging mechanism. The authors used a customized kernel on a real device, and the evaluation of the solution considers two malicious apps developed by the authors.

In [63] the authors characterize the response of an application in terms of a subset of system calls generated from untrusted activities in background when simulating apps with user interface events. They use an Android emulator for running experiments and their evaluation is based on two malicious samples belonging to the DroidDream malware family.

Authors in [58] propose an anomaly detection system for Android based on techniques analogous to those that can be used in a Linux kernel, based on network traffic, system calls, and file system logs.

The techniques presented in [37] are based on collecting and analyzing a set of system calls related to management of file, I/O and processes. A log collector records all system calls and filters events associated with the selected target application. A log analyzer matches activities with signatures described by predefined regular expressions, in the attempt of detecting a malicious activity. Signatures of information leakage are automatically generated using the smartphone IDs, e.g., phone number, SIM serial number, and Gmail accounts. The cited work used a physical device, with an Android 2.1 based modified ROM image. The evaluation phase considered 230 applications mostly downloaded from Google Play. The proposed method detected 37 applications which steal some kinds of personal sensitive data, 14 applications which execute exploit code and 13 destructive applications.

## 3. The proposed tool: VizMal

In this work, we focus on the visual evaluation of malware Android app through a visualization tool. Such tool aims for fast and precise individuation of malicious behaviors during the app execution. The tool takes as input an *execution trace* $t$ of an app and outputs an image consisting of a *sequence of colored boxes*, one box for each interval of $T$ seconds of $t$. An execution trace is a sequence of system call invocations, each invocation consisting only of an indication of the specific system call invoked without arguments.

The *color* of the box is related to the degree of maliciousness of the application behavior during the corresponding interval. The *shape* of the box (in particular, its *height*) is related to how active was the application in the corresponding second. The duration $T$ of the interval is a parameter: we set $T = 1\,\mathrm{s}$ which is a good trade-off between informativeness and easiness of comprehension.

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 LSTM: 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 image builder takes as input 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 syscall 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 subtrace (0 means no activity).

The image builder provides as output an image composed of an horizontal sequence (i.e., a row) of boxes, one for each element in the input sequence $L$. Boxes have the same width $w$ and are separated by a small empty gap for clarity. The height of the $i$-th box is $a_i w$, where $a_i$ is the activity level of the corresponding subtrace. The fill color of the $i$-th box is solid and determined based on $m_i$: 1. for $m_i = 0$, green; 2. for $m_i = 1$, red; 3. for $m_i \in (0, 1)$, the color corresponding to the point on a segment connecting green and red in the HSL color space whose distance from green is $m_i$ (distance normalized with respect to the segment length).

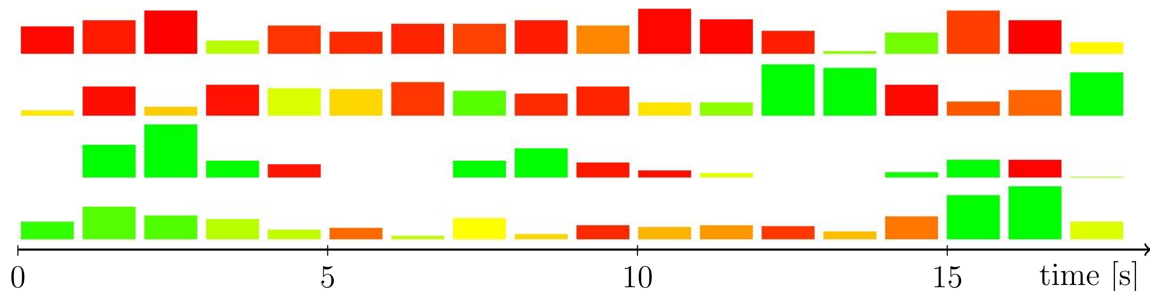



Figure 1: Examples of the images obtained by VizMal on 4 execution traces of malware apps: only the first 18 s are here depicted.

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 classifier is constructed in a preliminary *learning phase* that requires a set of labeled execution traces (benign vs malicious). In the *classification phase*, the trace classifier takes an unlabeled execution trace $t$ as input and provides a sequence $L$ of maliciousness and activity levels as output.

Execution traces are represented as sequences of *one-hot* vectors, as follows. Vectors are all the same size, equal to the number of possible system calls. Each vector element is associated with one of the possible system calls. Each system call invocation is represented by a vector with all elements set to 0 and only one element, the one associated with the invoked system call, set to 1.

The trace classifier consists of a *Long-Short Term Memory (LSTM)* neural network, a particular type of recurrent neural networks (RNN). RNN are a type of artificial neural network in which the processed information is not discarded, but recycled as an auxiliary weighted input. In this way, the network can mimic the human memory and maintain a kind of information persistence. However, traditional RNN can store information for few steps and tends to forget older information. LSTM tries to solve this problem introducing a special component in hidden recurrent layer. This component weights the recycled information so as to maintain what may be most useful in next steps. In this work, the network consists of a layer of 40 LSTM cells, followed by another layer of 40 cells that are fully connected to a layer with 10 neurons using Rectified Linear Units (ReLu) as activation function.

These neurons are finally fully connected to a single output neuron using softmax as activation function.

In the learning phase, the network takes a trace as input and provides a value $v \in [0, 1]$ as output. The network is trained to let the output value match the label of the input trace. In our experiments, the network has been trained for 300 epochs—i.e., each example of malware or non-malware trace has been seen 300 times by the network. The learning rate is dynamically varied during the learning with the Adam algorithm [40].

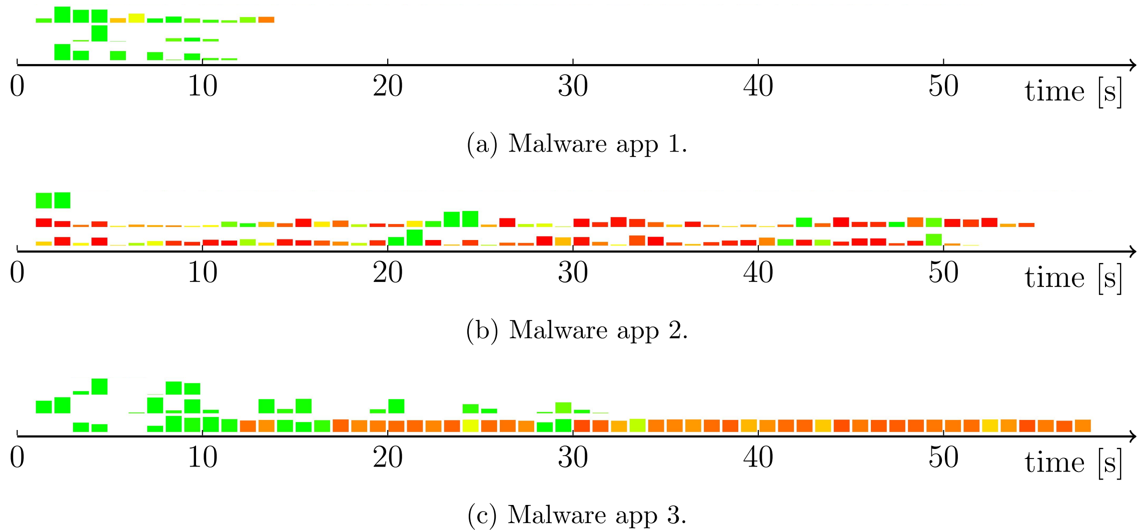
In the classification phase, the network takes a trace as input and provides a value $v \in [0, 1]$ as output. The output value is the maliciousness level of the corresponding input trace. In order to analyze each subtrace separately, in the classification phase we split each trace in subtraces and submit each subtrace $t_i$ to the network, thereby obtaining the corresponding maliciousness level $m_i$ for each $t_i$ The activity level of each subtrace $t_i$ is the normalized length of the subtrace, i.e., $a_i = \frac{|t_i|}{\max_i |t_i|}$.

## 4. Validation

We validated our proposal performing a set of experiments, i.e., to verify that VizMal may actually help the analyst in better understanding malware (and non-malware) apps behavior. It is important to emphasize how it is impossible to perform a quantitative evaluation of our proposal, since the labels are associated to the whole execution trace, not to the single sub-traces.

For this purpose, we used a dataset of 500 Android apps (a subset of those used in [14]), including 250 non-malware apps and 250 malware apps. We downloaded non-malware apps from the Google Play Store and malware apps from the Drebin dataset [7]. 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 [17] (i.e., by generating events with the monkeyrunner tool [3] and by recording the corresponding system call invocations).

In order to simulate the usage of VizMal to analyze new, unseen malware, we partitioned the dataset into two set of 450 and 50 apps. We trained our tool using the $450 \times 3$ traces corresponding to the former set and then applied the tool to the $50 \times 3$ traces of the latter. We repeated the procedure 10 times by varying the dataset partitioning and obtained consistent results. Several interesting observations may be made and we report here a subset of the images obtained in one




(a) Malware app 1.

(b) Malware app 2.

(c) Malware app 3.

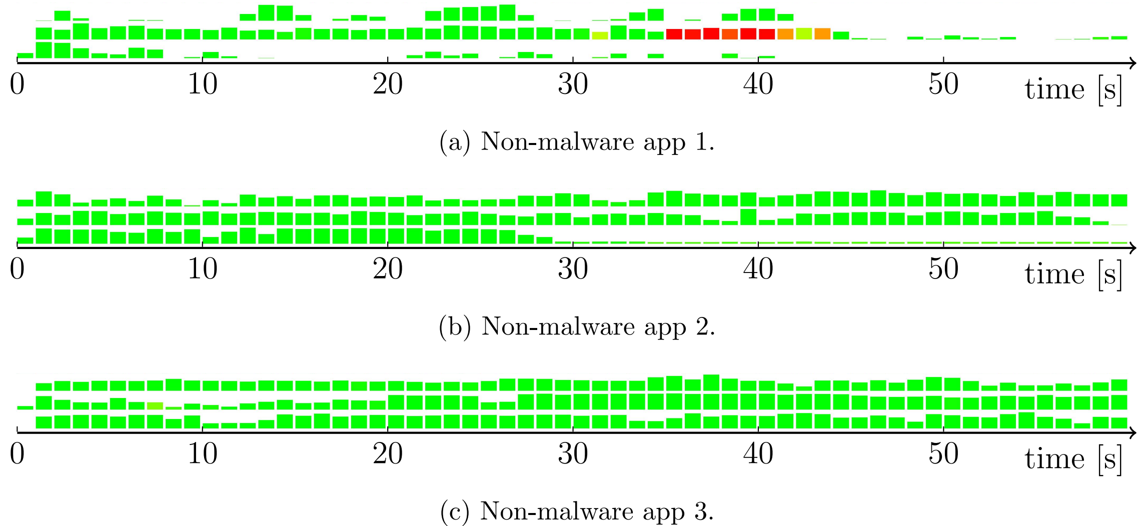
Figure 2: Traces from different malware apps represented by VizMal.

repetition.

Figure 2 shows the images obtained by Vizmal applied to the 3 traces of 3 malware apps. The different length of the visualized traces depends on the machinery used for stimulating the apps, which may generate events that can can stop the execution [17]. Red boxes indicate, with high confidence, intervals of malicious activity. Green boxes represent, with high confidence, intervals with no malicious activity. Yellow and orange boxes show intervals in which the execution is "borderline": these seconds are classified as non-malware (in yellow) and as malware (in orange) with a lower confidence. The figure also highlights varying activity levels during the execution: the height of the boxes clearly differentiates intervals with almost no activity (in terms of system call invocation) from intervals with a very high number of system call invocations.

Thanks to the VizMal tool, it is immediately observable when an app exhibits a general malicious behavior, when it behaves "normally" or "borderline". For instance, traces 2 and 3 of malware app 2 in Figure 2b, highlight a malware behavior; trace 1 and 2 of malware app 3 show benign activities; trace 1 of malware app 1 exhibits a "borderline" execution. As can be clearly understood, Vizmal allows an analyst to easily identify the intensity and the exact moment during which a malware behavior occurs.

Similar considerations can be made for images obtained with non-malware apps




(a) Non-malware app 1.

(b) Non-malware app 2.

(c) Non-malware app 3.

Figure 3: Traces from different non-malware apps represented by VizMal.

(Figure 3). The entire period of execution of 60 s is represented with a row of green boxes which means that the app does not present malicious behavior. It can also be seen that trace 2 of malware app 1 looks suspicious from the point of view of the sequences of the system calls. This can be an opportunity, for the analyst, to gain more insight into the execution trace or into the classification machinery.

It is important to note that VizMal applies to a single execution trace: the fact that a single trace may not be representative of the general behavior of an app in general is a problem orthogonal to the goal of VizMal. In this scenario, VizMal may be a valid tool for a faster analysis of several traces collected from the same app, perhaps in order to maximize the generality of findings, since it allows to quickly analyze a single execution trace.

### *4.1. Example of VizMal usage*

In order to better highlight the potentials of VizMal, we provide an example of how our tool cold be used for gaining more insights about an application. Figure 4 contains 4 trace of a malware from the ADRD family executed for 60 s. The ADRD malicious payload is able to use multiple infected devices to increase the site ranking for a given web site: its main purpose is the search engine manipulation and it focused on the search engine Baidu [44]. The ADRD malicious action starts when the infected device receive a call [69]. During the execution, we injected

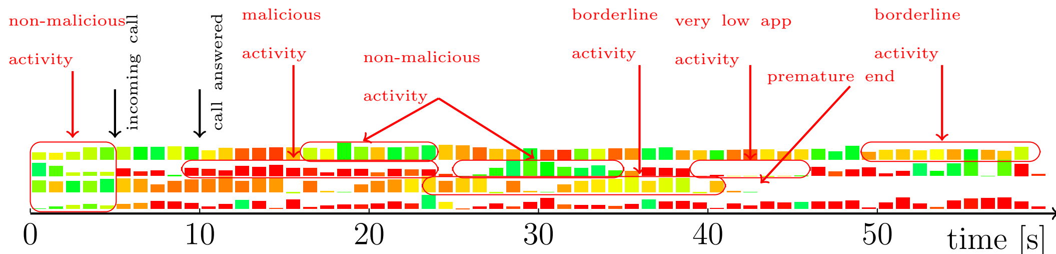



Figure 4: An example of VizMal applied to four execution traces of an app belonging to the ADRD malware family.

into the app a predefined sequence of events as described in [17]. A representative subset of such events is reported in the top part of figure.

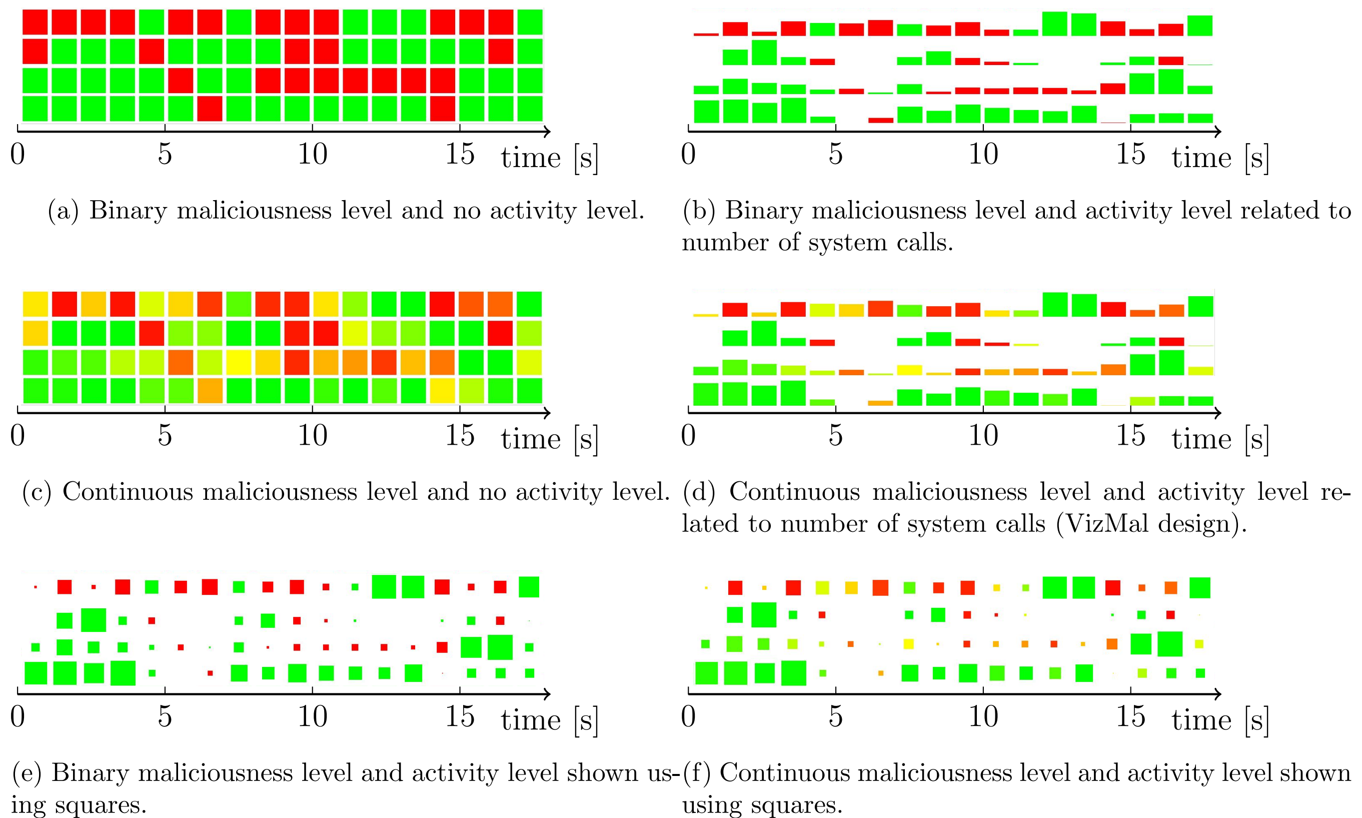
As can be seen from Figure 4, the initial behavior of the app is not malicious, probably because the malware waits for a trigger event. This view is reinforced by the fact that, according to VizMal outcome, the malicious activity starts when an event related to a call occurs. In fact, three of four execution traces show a potential malicious activity at the fifth second of activity, when an “incoming call” event is received by the app. Regarding the fourth execution trace, the malware activity does not star at the incoming call, but few seconds later, in conjunction with an “call answered” event. This behavior seems to be consistent with that described for the ADRD family in [69]: in fact, the ADRD malicious payload is triggered when the infected device receives a phone call [1].

Moreover, VizMal helps to understand that the behavior of the malware is not always malicious, but there are several seconds of execution during which the application behaves like a normal one. This could help an analyst to focus his effort in the malware analysis when certain events are generated, instead of taking trace of the whole execution of the app. Considering the third trace, another finding that emerges thanks to VizMal is the fact that in certain conditions the malware terminates its execution prematurely: this could be due to an intentional choice of the malware programmer or to an error in the malware code.

## 5. Alternative design options

### *5.1. Alternative image builders*

In this section we analyze some alternatives for the image builder we took into consideration for designing the final visualization step.




(a) Binary maliciousness level and no activity level.

(b) Binary maliciousness level and activity level related to number of system calls.

(c) Continuous maliciousness level and no activity level.

(d) Continuous maliciousness level and activity level related to number of system calls (VizMal design).

(e) Binary maliciousness level and activity level shown using squares.

(f) Continuous maliciousness level and activity level shown using squares.

Figure 5: Examples of visualization for 3 malware traces and 1 non-malware trace (bottom row).

In the first and simpler solution, boxes convey only a binary indication for the maliciousness level (either green or red) and no indication for the activity level. The result is shown in Figure 5a for 3 execution traces of malware apps (top) and one trace of a non-malware app (bottom). The resulting information is too coarse grained to be useful for the analyst, as the bottom row (benign trace) is quite similar to the second row (malware trace).

For this reason, we decided to introduce an indication of the activity level—i.e., the number of system calls executed during each subtrace. An example of this visualization is shown in Figure 5b, for the same traces of Figure 5a. In this new visualization, the difference between the benign trace and malware traces is much more evident: it is much easier to notice that in the trusted app the activity classified as suspicious is much lower, indeed, fewer system calls were executed during the time interval if compared to the malware apps.

A different approach for improving Figure 5a consists in highlighting the maliciousness of a subtrace with a continuous value (as opposed to a binary one) while omitting any indication related to the activity level. We encoded the confidence of the classification using the fill color of the boxes, i.e., based on the $m_i$ value as described in Section 3.1. The result can be seen in Figure 5c. The resulting visualization is certainly more informative and useful than the one in Figure 5a, but the activity level seems to be a crucial component to be presented to the analyst.

The design finally adopted for VizMal is shown in Figure 5d, for the same traces as the other figures. It can be seen that the visualization conveys a much richer and more useful amount of information than any of the considered alternatives. It can also be observed that, with this visualization, the trace of the non-malware app does not exhibit any subtrace that looks highly suspicious. With the other visualizations, instead, the analyst could be encouraged to devote precious resources in analyzing some subtraces of the non-malware app.

For completeness of discussion, we mention a different design option that we explored for visualizing the activity level. In this variant the activity is shown with squares whose size is proportional to the activity level Figure 5e. We decided to not use this approach because the maliciousness level in smallest boxes is barely visible—when the maliciousness level is continuous the problem is even more evident. We also report in Figure 5f the variant with the continuous maliciousness level.

### *5.2. Alternative trace classifiers*

In order to explore different options for the trace classifier, we considered 3 other algorithms able to work in the considered scenario. We remark that the key problem consists in the fact that labels (benign vs malware) are associated with full traces while we need to associate a separate label with each single subtrace, depending on the degree of maliciousness of that subtrace. In this respect, the main motivation for our usage of an LSTM neural network is that we can train LSTM on a full trace and then use the trained classifier for labelling traces of arbitrary length, thus even subtraces of a longer trace.

Another approach which can fit our scenario is the *Multiple Instance Learning (MIL)* framework [22, 70]. This is a form of weakly-supervised learning in which the learning set is composed by instances grouped together and, instead of labelling each instance, the label is associated with a group.

We experimented with different variants of MIL approach: sMIL [13], miSVM [6], and an ad hoc variant of "single instance" SVM (SIL-SVM) which we modified in order to act as a MIL framework. For sMIL and miSVM, a label is associated with an entire trace with the following semantic: a malware label means that at least one subtrace is malware; a non-malware label means that all the subtraces are non-malware. For SIL-SVM, a malware label means that all subtraces are malware app and vice versa.

Unlike the LSTM approach, the MIL-based ones require a *feature extraction* procedure in order to process a subsequence. We used the same feature proposed in [17]: frequencies of n-grams of the system calls occurred in the trace. The feature extraction procedure occurs as follows:

(i) we split a trace $t$ in a sequence $\{t_1, t_2, \dots\}$ of subtraces, where each subtrace lasting exactly $T$ seconds; (ii) we considered subsequences (n-grams) of at most 5 consecutive system call invocations; (iii) finally, we counted the number $o(t_i, g)$ of each n-gram $g$ in each subtrace $t_i$.

In order to assess the 3 variants and to compare them with LSTM approach, we performed the following procedure:

1. we divided the dataset of $3 \times 500 + 3 \times 500$ 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 all the classifiers on the learning set;

3. we applied the trained classifiers on the traces in the testing set.

The above procedure has been repeated with a 10-fold cross-validation. For each repetition we measured the performance of the classifiers in terms of False Positive Rate (FPR) and False Negative Rate (FNR). Since all the considered MIL variants base on SVM, for a fair comparison we used the linear kernel and $C = 1$ for all.

Before discussing the results, we emphasize that while it is fair to consider a false positive as an error (i.e., a subtrace of a non-malware app classified as a malware), it is not fair to consider a false negative as an error (i.e., a subtrace of a malware app classified as a non-malware). The reason is because a malware app may exhibit a legitimate behavior for several seconds during its execution. We also remark that (a) our experimentation was not aimed at performing a comparison among MIL frameworks—the interested reader may refer to [55]—and (b) the shown figures

Table 1: FPR and FNR for LSTM and the three considered variants of MIL classifiers.

| | Natural threshold | | Near EER | |
|---|---|---|---|---|
| Classifier | FPR | FNR | FPR | FNR |
| SIL-SVM | 0.758 | 0.113 | 0.402 | 0.385 |
| miSVM | 0.264 | 0.424 | - | - |
| sMIL | 0.098 | 0.693 | 0.308 | 0.364 |
| LSTM | 0.098 | 0.551 | 0.244 | 0.260 |

should not be intended as representative of the accuracy of malware detection for the considered approaches.

Table 1 presents the results, averaged across the 10 repetitions. The most important result is that LSTM outperforms all the MIL approaches in terms of false positive rate. This result corroborates our design choice of basing VizMal on an LSTM network and is a very interesting outcome, because, as pointed out above, FPR is the most important index. Concerning the MIL variants, miSVM outperforms both SIL-SVM and sMIL.

The Table 1 also reports[1] the FPR and FNR values at Equal Error Rate (EER) point. The performance indices of the trace classifier can vary in accordance to a threshold. This indication can be interesting in the perspective of an interactive tool in which the analyst can vary the classifier threshold in order to modify the color of the boxes.

## 6. User-based validation

The motivation for VizMal is to support dynamic analysis and reduce the effort in individuating the event triggering the malicious behaviour. The degree to which this goal can be achieved in practice is determined also by the usability of the proposed tool. In order to assess the usability and, hence, the usefulness of VizMal, we performed a user study.

The study consisted of a questionnaire which we gave to a set of users familiar with the concept of malware analysis, but to different degrees. The questionnaire

[1] The EER values for miSVM are missing due to a limitation of the implementation.

was composed of four parts: presentation of the tool, simulated usage (two parts), perceived usefulness.

The first part introduced VizMal with a detailed description of its functionality and behaviour, with particular attention to the appearance and the meaning of its output. The second one presented the graphical visualizations obtained by applying VizMal to two applications, each one with three execution traces, and asked if the applications appeared as malware or trusted application. In the third part, the questionnaire showed an execution trace lasting 60 seconds. This trace was decorated with the events injected during the execution (similarly to Figure 4: the user can find an arrow annotated with the name of the event in correspondence of the second in which it has occurred. Then, we asked to the participants to identify which event triggered the malicious routine and how long the malware activity lasted. Finally, the last part of the questionnaire proposed four qualitative questions about the perception the user had about the tool. These questions concerned about (a) how immediate was to identify the event activating the malicious behaviour, (b) how immediate was to identify the duration of a malicious behaviour, (c) whether VizMal can be useful as a tool to reduce the work of the malware analyst, and (d) whether it is easy to identify the behavior labeled by VizMal as malicious. Each of these questions required a value between 1 and 10 as a response, where the lowest value means a fully negative answer (e.g., not immediate) and the highest means a fully positive answer (e.g., very immediate). We made the questionnaire and the complete set of answers publicly available [2].

### *6.1. Questionnaire results*

The survey was sent to a set of people familiar with the concept of malware analysis, including undergraduate, graduate, and PhD students of the Università del Molise, Italy (Computer Science and Software systems programs), and the Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche (CNR), Italy. We collected 33 filled questionnaires. The set of respondents was composed by 25 bachelor, 4 master, and 4 PhD students.

Regarding the second part, concerning the identification of malware and trusted applications, all users correctly identified the malicious and trusted application. With respect to the third part, 32 over 33 participants correctly answered the question about the system event triggering the malicious behaviour (i.e., battery status discharging). One participant indicated two events as possible triggers, one being the correct one. These results appear to validate our claim that VizMal is helpful in identifying the event activating the malicious behavior.

Concerning the duration of the malicious activity, despite the users were in principle allowed to input any integer value between 0 and 60, they responded with values within a limited range, between 6 and 10 seconds. The majority of the replies indicate 7 seconds (i.e., the correct answer) as the duration of the malware activity (13 over 33, 39% of the replies), followed by 10 seconds (9 users, with 27% of the replies). The limited disagreement among users' answers show how VizMal can facilitate the potentially hard task of estimating the duration of the malicious behavior in a rather consistent way.

The results of the last part of the questionnaire are summarized in Figure 6. The majority of the responses agreed that VizMal helps in identifying the event which triggers the malicious behaviour (91% of the replies with a score greater than 6). In particular, 85% of the replies assigned a values equal or greater than 8. This results is very promising, since one of the contributions of VizMal is to help analyst in individuating when the applications begins to act as a malware. Concerning the second statement, about the ease of identifying the duration of malware activity, again the majority of users agreed in finding VizMal useful. Anyhow, 2 users gave a score equal to 4 and the majority of votes are limited to 8 (32% of the cases) and 9 (38%). The third statement asked if VizMal can allow an analyst to reduce his own work. Respondents mostly agreed, with the 90% of the replies with score greater than 7. The last statement explored the perception of the user in the ease of finding the malicious behaviour using VizMal. The users clearly felt that our tool permits to easily find this portion of the execution, since the majority of them assigned a score equal or greater than 8 (75%) and the reaming gave a score of 7 (25%). We believe these results are highly encouraging. Indeed, from this questionnaire, it is clear that users agreed in finding VizMal a valid tool for reducing the work required in dynamic analysis.

## 7. Concluding remarks

In this paper we presented a tool, called VizMal, for displaying the results of a dynamic malware analysis of Android applications. VizMal visualizes an execution trace of an Android application as a row of colored boxes. Each box corresponds to one second of execution and the color and the shape of the box give information about the maliciousness and the activity levels of the app during the corresponding second of execution. Internally, VizMal exploits the proven ability of LSTM neural network to cope with long sequence of symbols—i.e., long sequence of system calls. We think that VizMal may be a useful and valuable tool for Android

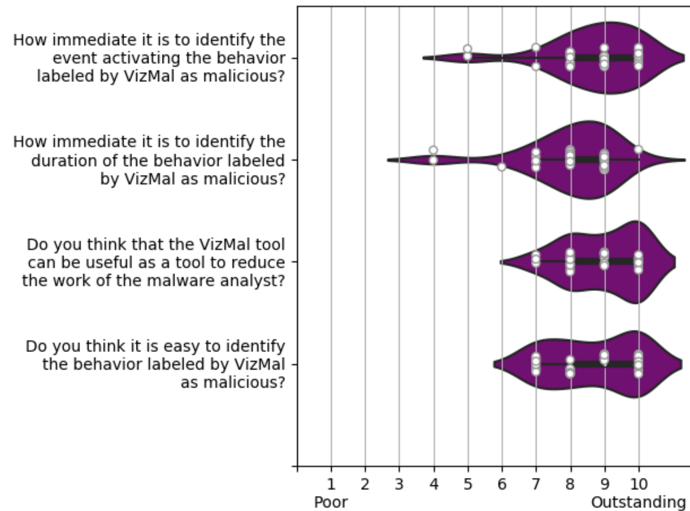



Figure 6: Violin plot indicating the responses to the 4 statements of the fourth part of the questionnaire. The purple shape represents the overall distribution of the results. White points are the responses.

malware analysts, since it can allow them to better understand the behavior of malware application. Although our tool certainly has to be investigated further, we believe that VizMal may be an interactive tool for malware analysis, especially for investigating the relation between the maliciousness activity and the events generated by system or user interaction. We plan to extend the VizMal analysis also for malicious behaviour related to desktop platforms by adding more graphical representations.

## Acknowledgments

This work has been partially supported by MIUR - SecureOpenNets and EU SPARTA and CyberSANE projects and WEBREPUTO POR FESR 2014-2020

## References

[1] Symantec. `https://www.symantec.com/security-center/writeup/2011-021514-4954-99`, last visit 2 November 2019.

[2] Machine learning lab. https://machinelearning.inginf.units.it/data-and-tools/visualizing-the-outcome-of-dynamic-analysis-of-android-malware-with-vizmal, last visit 7 November 2019.

[3] monkeyrunner. https://developer.android.com/studio/test/monkeyrunner, last visit 7 November 2019.

[4] Y. Aafer, W. Du, and H. Yin. 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, 2013.

[5] O. S. Adebayo and N. A. Aziz. The trend of mobile malwares and effective detection techniques. In *Multigenerational Online Behavior and Media Use: Concepts, Methodologies, Tools, and Applications*, pages 668–682. IGI Global, 2019.

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

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

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

[9] A. Bacci, F. Martinelli, E. Medvet, and F. Mercaldo. Vizmal: A visualization tool for analyzing the behavior of Android malware. In *International Conference on Information Systems Security and Privacy*, pages 517–525, 2018.

[10] U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[11] M. L. Bernardi, M. Cimitile, D. Distante, F. Martinelli, and F. Mercaldo. Dynamic malware detection and phylogeny analysis using process mining. *International Journal of Information Security*, pages 1–28, 2018.

[12] T. Blasing, A.-D. Schmidt, L. Batyuk, S. A. Camtepe, and S. Albayrak. An Android application sandbox system for suspicious software detection. In *Proceedings of 5th International Conference on Malicious and Unwanted Software*, 2010.

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

[14] G. Canfora, A. De Lorenzo, E. Medvet, F. Mercaldo, and C. A. Visaggio. 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, 2015.

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

[16] G. Canfora, F. Martinelli, F. Mercaldo, V. Nardone, A. Santone, and C. A. Visaggio. LEILA: formaL tool for idEntifying mobIle maLicious behAviour. *IEEE Transactions on Software Engineering*, 2018.

[17] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio. 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, 2015.

[18] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio. 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, 2016.

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

[20] G. Canfora, F. Mercaldo, and C. A. Visaggio. A classifier of malicious Android applications. In *2013 International Conference on Availability, Reliability and Security*, pages 607–614. IEEE, 2013.

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

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

[23] A. Cimitile, F. Martinelli, F. Mercaldo, V. Nardone, and A. Santone. 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, 2017.

[24] A. Cimitile, F. Mercaldo, V. Nardone, A. Santone, and C. A. Visaggio. Talos: no more ransomware victims with formal methods. *International Journal of Information Security*, pages 1–20, 2017.

[25] M. Dalla Preda and F. Maggi. 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, 2017.

[26] B. Dixon, Y. Jiang, A. Jaiantilal, and S. Mishra. Location based power analysis to detect malicious code in smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 2011.

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

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

[29] M. Faiella, A. La Marra, F. Martinelli, F. Mercaldo, A. Saracino, and M. Sheikhalishahi. A distributed framework for collaborative and dynamic analysis of Android malware. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 321–328. IEEE, 2017.

[30] F. Fasano, F. Martinelli, F. Mercaldo, and A. Santone. Cascade learning for mobile malware families detection through quality and Android metrics. In *2019 International Joint Conference on Neural Networks, IJCNN 2019*, volume 2019, pages 1–10. IEEE, 2019.

[31] F. Fasano, F. Martinelli, F. Mercaldo, and A. Santone. Energy consumption metrics for mobile device dynamic malware detection. *Procedia Computer Science*, 159:1045–1052, 2019.

[32] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, and S. Furnell. Androdialysis: analysis of Android intent effectiveness in malware detection. *Computers & Security*, 65:121–134, 2017.

[33] A. Ferrante, E. Medvet, F. Mercaldo, J. Milosevic, and C. A. Visaggio. 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, 2016.

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

[35] G. Geogen and E. Poovammal. Mobile malware. In *Securing the Internet of Things: Concepts, Methodologies, Tools, and Applications*, pages 92–108. IGI Global, 2020.

[36] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with LSTM. 1999.

[37] T. Isohara, K. Takemori, and A. Kubota. Kernel-based behavior analysis for Android malware detection. In *Proceedings of Seventh International Conference on Computational Intelligence and Security, pp. 1011-1015*, 2011.

[38] Y.-s. Jeong, H.-t. Lee, S.-j. Cho, S. Han, and M. Park. A kernel-based monitoring approach for analyzing malicious behavior on Android. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1737–1738. ACM, 2014.

[39] H. Kim, J. Smith, and K. G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, 2008.

[40] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[41] M. Lindorfer, M. Neugschwandtner, and C. Platzer. 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, 2015.

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

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

[44] F. Martinelli, F. Mercaldo, V. Nardone, and A. Santone. How discover a malware using model checking. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 902–904. ACM, 2017.

[45] F. Martinelli, F. Mercaldo, V. Nardone, A. Santone, A. K. Sangaiah, and A. Cimitile. Evaluating model checking for cyber threats code obfuscation identification. *Journal of Parallel and Distributed Computing*, 119:203–218, 2018.

[46] F. Martinelli, F. Mercaldo, and A. Saracino. 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, 2017.

[47] E. Medvet and F. Mercaldo. 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, 2016.

[48] F. Mercaldo, V. Nardone, and A. Santone. Ransomware inside out. In *Availability, Reliability and Security (ARES), 2016 11th International Conference on*, pages 628–637. IEEE, 2016.

[49] F. Mercaldo, V. Nardone, A. Santone, and C. A. Visaggio. Hey malware, i can find you! In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2016 IEEE 25th International Conference on*, pages 261–262. IEEE, 2016.

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

[51] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. 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*, 2013.

[52] M. Polino, A. Scorti, F. Maggi, and S. Zanero. Jackdaw: Towards automatic reverse engineering of large datasets of binaries. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 121–143. Springer, 2015.

[53] A. Qamar, A. Karim, and V. Chang. Mobile malware attacks: Review, taxonomy & future directions. *Future Generation Computer Systems*, 97:887–909, 2019.

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

[55] S. Ray and M. Craven. 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, 2005. ACM.

[56] J. Rhee, R. Riley, D. Xu, and X. Jiang. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In *International Workshop on Recent Advances in Intrusion Detection*, pages 178–197. Springer, 2010.

[57] A. Santone and G. Vaglini. Abstract reduction in directed model checking ccs processes. *Acta Informatica*, 49(5):313–341, 2012.

[58] A.-D. Schmidt, H.-G. Schmidt, J. Clausen, K. A. Yuksel, O. Kiraz, A. Camtepe, and S. Albayrak. Enhancing security of linux-based Android devices. In *Proceedings of 15th International Linux Kongress*, 2008.

[59] A. Shabtai, U. Kanonov, and Y. Elovici. Intrusion detection for mobile devices using the knowledge-based, temporal abstraction method. In *Journal of Systems and Software, 83(8):1524–1537*, 2010.

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

[61] T. Shibahara, T. Yagi, M. Akiyama, D. Chiba, and K. Hato. Efficient dynamic malware analysis for collecting http requests using deep learning. *IEICE Transactions on Information and Systems*, 102(4):725–736, 2019.

[62] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic reconstruction of Android malware behaviors. In *In Proceedings of the Network and Distributed System Security Symposium*, 2015.

[63] F. Tchakounté and P. Dayang. System calls analysis of malwares on Android. In *International Journal of Science and Tecnology (IJST) Volume, 2 No. 9*, 2013.

[64] L. Wang, B. Wang, J. Liu, Q. Miao, and J. Zhang. Cuckoo-based malware dynamic analysis. *International Journal of Performability Engineering*, 15(3), 2019.

[65] M. Wazid, S. Zeadally, and A. K. Das. Mobile banking: evolution and threats: malware threats and security solutions. *IEEE Consumer Electronics Magazine*, 8(2):56–60, 2019.

[66] F. Wei, S. Roy, X. Ou, et al. 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, 2014.

[67] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, 5(2):32–39, 2007.

[68] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah. Android malware detection based on system call sequences and LSTM. *Multimedia Tools and Applications*, pages 1–21, 2017.

[69] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.

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