GroDDViewer: Dynamic Dual View of Android Malware

Jean-François Lalande, Mathieu Simon, and Valérie Viet Triem Tong

CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA, Rennes, France
{firstname.lastname}@inria.fr

Abstract. Understanding an Android malware is a difficult task that requires strong skills in reverse engineering. Few tools exist except the well know IDA and Ghidra tools that are more focused on the analysis of binaries. In the Android world, understanding a malware requires to analyze the bytecode of the application, possibly obfuscated or hidden in a benign application that have been modified. At execution time, the malware can download new payloads, compromise the smartphone, install new apps. We believe that a security analyst would appreciate to visualize and replay an execution of an Android malware. In particular, an analysis that bridge the gap between the bytecode and the events occurring during the execution would help to understand the malware behavior. In this article, we propose GroDDViewer the first tool offering a dual view of the execution of an Android malware. The first view represents the execution at operating system level through the representation of all information flows between files, processes and sockets. The second view represents what happened in the code of the application, during its execution. The benefit of this visualization tool is illustrated on a ransomeware sample. Future works concern the evaluation of the tool with a panel of users on a benchmark of malware samples.

Keywords: malware, visualization

1 Introduction

Security researchers have different goals when working on Android malware analysis. Faruki et al. have discussed these goals and the associated methodologies [5]. Most of contributions try to decide if an application is a malware or not. Few works try to address the problem of understanding the behavior of a malware application. Nevertheless, such an activity is an important task for security analysts of companies or government agencies that are involved in cyber security. Analyzing and understanding Android malware can have multiple goals. Most of the time it consists in locating a payload, triggering it, for example if it is encrypted. By observing the actions of the malware, the analyst should be able to classify a sample as a locker, RAT, ransomeware, etc. If the application has been piggybacked [9], the analyst should fin out the malicious code. Then, he has to understand what the malicious code is doing, when executed, and we believe

that for these tasks, the security analyst need to be helped by tools, especially visualization tools.

A lot of approaches are based on static analysis but well known contributions such as CopperDroid [14], CrowDroid [4], DroidScope [19], Harvester [3] have focused on extracting malware information from an execution. As mentioned by Faruki et al., such approaches have to face to the difficulty of being sure that the malware has been successfully executed. Thus, new approaches [1, 6] focused on the particular problem of helping the execution of malware that wait for particular conditions to occur.

Nevertheless, all these dynamic approaches focus on *how to get* data from an execution (system calls, variable values, network operations, etc.) but not on *how to display* the captured data for the security analyst. Most of the time, online platforms that propose an analysis report give basic textual information about a sample, like virustotal or Andrubis [18]. Such tools can give aggregate view of a huge amount of malware samples analyzed, like one million analysis of Andrubis [10]. Aggregate views are useless for the security analyst that needs to gain information of a particular sample, especially if this sample is a new discovered one that have never been analyzed before.

In this article, we propose a new visualization approach, GroDDViewer, for helping a malware analyst to gain information about the execution of a malware sample. GroDDViewer gathers a static approach and a dynamic analysis. This way, GroDDViewer offers a dual view of the execution of the malware: one view dedicated to the representation of the attack by all the information flows generated at operating system level between processes, files and sockets and a second view dedicated to the representation of the executed malicious code. These two views can be manipulated by the analyst and can focus on precise intervals of time. Additionally, GroDDViewer offers a replay feature to animate the two views and *see* the malware operating and executing itself. GroDDViewer is implemented as a webpage in order to be easily accessible from any platform in a web browser.

The rest of the article is structured as follows. Section 2 presents the approaches related to the visualization of Android malware. Section 3 briefly explains how are collected the data from a malware execution before moving on Section 4 that presents the visualization interface of GroDDViewer. We illustrate our tool on a real malware use case in Section 5. Finally, Section 6 concludes the paper.

2 Related works

Visualizing malware can rely on static or dynamic approaches. Previous works use static analysis to extract malware features that can be used to classify, browse malware families or study one particular malware [16]. On the contrary, few papers focus on the visualization of dynamic analysis. This is surprising, as malware analysts need to collect information from executions, for example API

and kernels calls [17]. A complementary approach is to monitor the network during an analysis, which can give good insight of malware activities [20].

We found several approaches that focus on the analysis of one single malware and capture dynamic data and propose visualization results that have similarities with our approach. Trinius et al. [15] propose to use treemaps to visualize system calls and treegraphs to represents the system commands during the time of execution. This work is similar to our approach as is a sort of dual view of a malware (system and command levels) with a dynamic view that helps the user to get what is happening over time. Another work that have similar ideas is the paper of Grégio et al. [7] that represent system calls using a Jung graph. In [13], Quist et al. introduce the visualization of the control flow of the program for executable malware. This approach produces very large graphs but helps to isolate loops, and especially unpacking stages which is of primary interest for x86 malware. Compared to our approach, we intend to use the control flow graph to link observations to the reversed source code of the malware. Thus, we need to have more readability on such kind of representation.

Visual analysis can be used to classify or recognize malware. In [11], authors use visual similarities of malware's image to discover relationships between malware. In [12], a graphical overview of the similarities of Android malware's code help to identity the shell code shared by different malware samples. These approaches have a different goal because they help to understand the evolution of a family of malware or multiple samples.

When dealing with a unique malware, well known online platforms give very basic information, mainly in a text based way. The most developed source of information are the blogs web pages that give precise insight for a particular sample. Such analysts use virtualized emulators or real smartphones to execute the malware and can be helped by uncompilers or debuggers like the well known IDA software. Nevertheless, such tools have no advanced display capabilities when a malware operates million of system calls, creates hundreds of files and have thousands of Java classes to understand. The particular nature of Android applications and the way the malware are implemented, as a repackaged benign application where malicious code has been added, pushed us to develop a new visualization interface.

Additionally, all the cited approaches are related to the visualization of x86 malware and do not focus on the particularities of Android malware (except for Paturi et al. [12]). Thus, we believe that this paper is the first to propose a visualization for Android malware combining the view of the code and the operating system events captured during an execution.

3 Material collection

GroDDViewer gathers static and dynamic analysis in order to offer a representation of the attack itself and the malicious code that has been executed during the attack.

First the malicious behavior is captured by AndroBlare [2], that monitors flows of information at operating system level. AndroBlare intercepts system calls responsible of information flows between files, sockets or processes which enable to observe the malware from the operating system point of view. AndroBlare relies on tainting techniques : the malware APK file is tainted with a mark and each process or object of the system can obtain the mark if a system call generates an information flow from a marked process/object. During the execution, all the interactions that happened between the process created from the APK file and the system are collected in a log. These interactions are process creation, file creation, socket interactions. We also collect the state of the user files before and after an execution in order to be able to show what happened to the files.

The attack is triggered by GroddDroid 2 [1] a framework that detects suspicious code and controls multiple executions of a malware in order to force the execution of code identified as suspicious. GroddDroid 2 instruments the bytecode to be able to trace the execution of all branches of the control flow. Then, it executes and stimulates the malware in a real smartphone and audits the executed branches. If the suspicious code is not reached, GroddDroid 2 changes branch conditions in order to push the execution towards the malicious code. During such multiple executions, we collect the name and the time of the executed branches in order to be able to give a representation of the executed code at method level, as described later in Section 4.

4 Visualizing malware execution

4.1 Overview

GroDDViewer offers a dual view of a malware execution: a view of all the information flows at operating system level and a view of the executed malicious bytecode. As shown in Figure 1 and 2, four main components explain the malware execution:

1. **System Flow Graph**: represents all the information flows induced by the malware execution that occurred at system level;
2. **Interactions frequency**: represents the number of information flow events over time;
3. **Method Control Flow and Bytecode View**: represents the control flow of method calls.
4. **User interface navigation**: represents what is seen from the user perspective, if any, and the events to go from one screen to another one.

Dynamic interactions of the user with these graphical elements provide additional information. For example, the user can click to get additional information such as a file modification or the bytecode source. The selection of time intervals provides a zoom capability on a specific period of time. The replay feature animates the graphs in order to replay events at operating system and bytecode levels. All these features are described in the next sections.

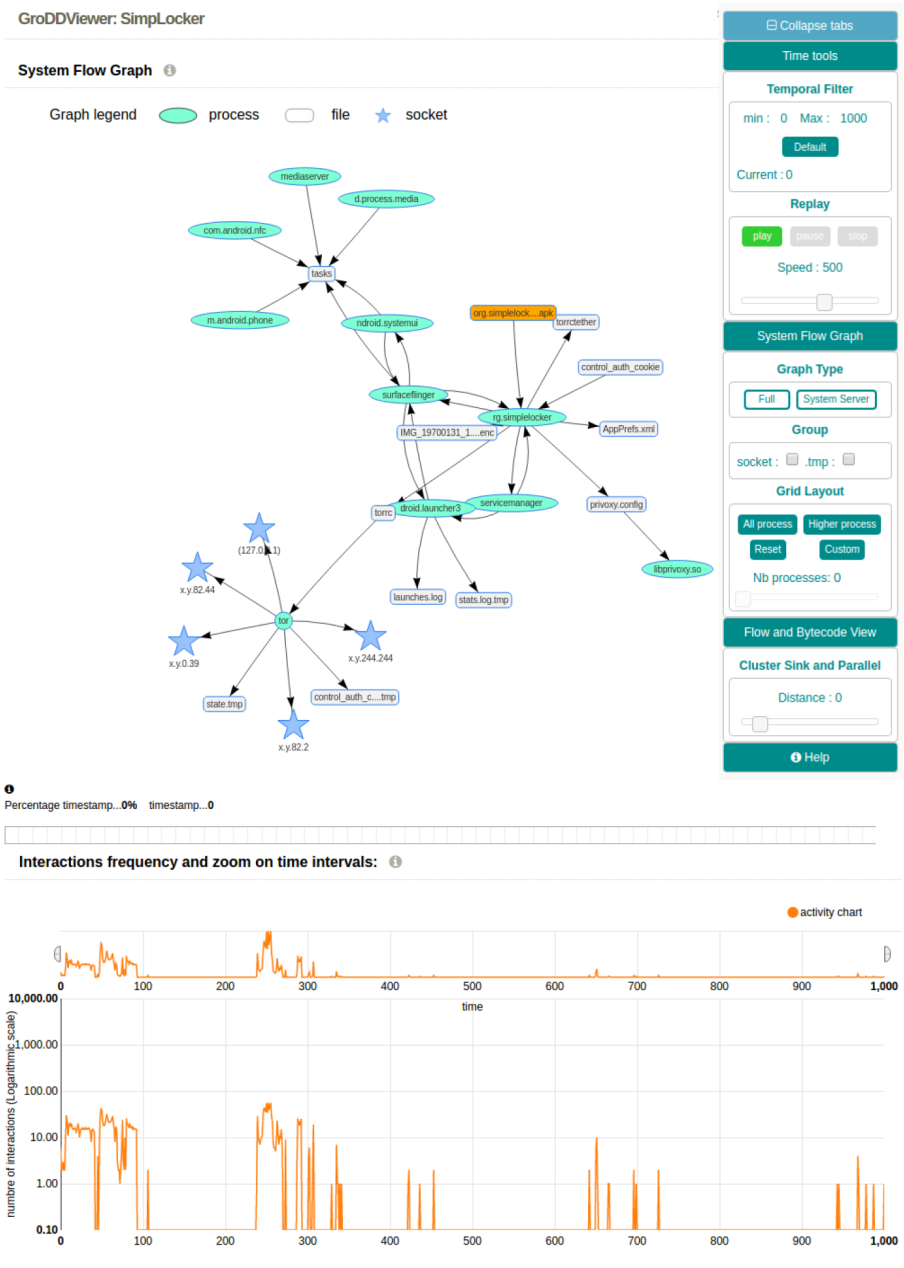

Fig. 1. Overview of GroDDViewer (part 1)

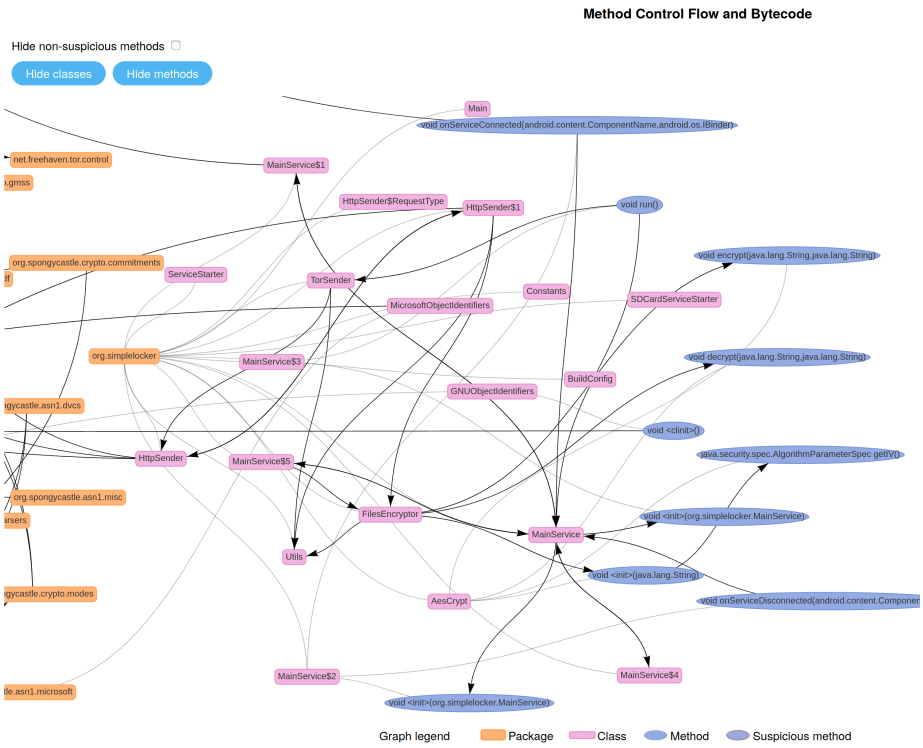

Fig. 2. Overview of GroDDViewer (part 2)

4.2 System Flow Graph

Information flows between objects of the operating system represent how the malware contaminates the operating system from the APK file (upper part of Figure 1). Each edge of the graph may appear multiple times as system calls can be triggered often by the process, for example when writing a file. We record the timestamps of each occurrence which enables to replay the interactions.

A node of the graph can be a process, a socket, or a file. When clicking on a file, the difference of content is displayed between the initial state and final state of the experiment, if the file is a text file. It allows to follow the content modified or created by a malware. If the malware just read information, the edges show a transition from the file to a process.

The toolbar provides additional functionalities to manipulate the System Flow Graph. First, additional nodes can be displayed. The *Full* graph option shows the possible duplicate process nodes. It corresponds to the execution of multiple independent processes that have the same name. The *System Server* graph option shows the subgraph of the System Server process and all connected other processes that have been contaminated by the mark through System

Server. As System Server is the central process that delivers Android Services and may asks to other Android process some data, the size of this subgraph can be very large if the malware accesses frequently the Android API. Thus, masking this part of the graph helps to visualize the processes that are accessed by the malware but it may be necessary to reactivate it to learn what the malware tries to access. Second, nodes that have similar extensions can be grouped. It allows to reduce the graph when a malware generates a lot of similar files, for example writes log files or accesses multiple sockets.

Finally, the layout of the processes can be controlled using the *Grid Layout* option. It forces the placement of all, higher or a custom number of processes on a grid. This tool helps to browse the graph when the number of nodes is large.

4.3 Interactions frequency

In bottom part of Figure 1, a frequency graph displays the number of events occurring for information flows at kernel level. Because a simple Java operation can generate a large number of system calls, the number of flows in few milliseconds can be very high. Thus, we discretize the time of experiments in an interval $[0, 1000]$ and we display the number of events on a logarithmic y axis.

The interaction frequency graph also intends to be used for zooming on a precise time interval. Indeed, some malware actions can be concentrated in a particular portion of time: the user selects a new time interval in $[0, 1000]$ on the upperpart of the interaction frequency graph. A new selection of an interval $[x, y]$ has two effects. First, the lower orange graph is updated accordingly. Second, the System Flow graph is updated to display the processes, files and sockets involved during $[x, y]$. This functionality is particularly useful for understanding what the malware is doing on a particular period of time, or where the user shows a pick of activity on the Interaction frequency graph.

4.4 Control flow and bytecode views

The dual view of the System Flow Graph is the Method call graph that represents the control flow between methods. We could have displayed the entire control flow, i.e. by representing the control flow of the inside of a method, but the graph would have become difficult to understand. Thus, we define the nodes as methods and the edges represents explicit calls of methods. This way, we obtain a graphic representation of the code of the malware, where each node can be clicked to display the bytecode source. A path, in such a graph, is a possible nested suite of method calls until a return statement unstack the last call.

As shown in Figure 2, in order to help the user to browse the graph, we give the possibility to fold/unfold the methods (blue nodes) by packages (orange) and classes (pink). Suspicious classes have a red border and help the user to focus on suspicious methods. Each node of the call graph can be clicked. GroDDViewer displays the bytecode in a popup window, as shown in Figure 3. This way, the user can analyze the suspicious bytecode and follow the malware developer logic.

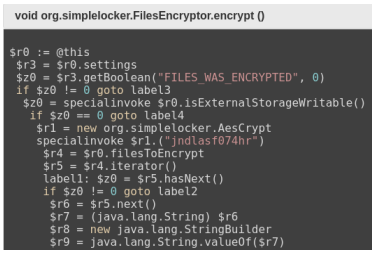

Fig. 3. Bytecode visualization

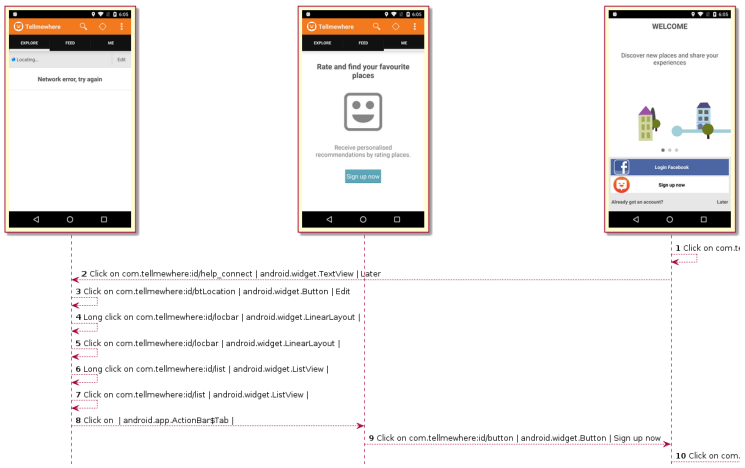

Fig. 4. Automaton of the navigated screens

4.5 User interface navigation

GroDDViewer also displays the different screens of the application that appeared during an execution, as shown in Figure **??**. These screenshots are represented as automaton where transition are labeled with the simulated user interaction.

4.6 Dynamic replay

As the collected data come from an execution of the malware, we also record the timestamps associated to all events: the dates of the observed flows of the System Flow Graph and the dates of the branches of the control flow graph of the bytecode. The collected timestamps are extracted from the kernel (for the System Flow Graph) or from the Android `logcat` command when the malware bytecode is executed. Thus, we have to synchronize the two sources of timestamp to be able to replay events with a precision acceptable from the user perspective.

The Replay feature, located in the Time tools group of the toolbar, replay all events in a dual manner: the System Flow Graph events are animated synchronously with the Method Control Flow graph. This animation helps to see simultaneously the operation at system level, for exemple file creation or socket communication, while the methods of the bytecode are called. It helps to identify the nature of the methods from the nature of the performed action in the system, as illustrated in the use case in Section 5.

5 Use case

We have chosen to study a ransomware called SimpleLocker[1] from the Kharon dataset [8] to present an example of use case of GroDDViewer. SimpleLocker is a ransomware that encrypts the user files before asking for a ransom to the user. If the user pays the ransom, the attacker may trigger the unencryption process using the Tor network.

5.1 Static analysis

When displaying the GroDDViewer page for the SimpleLocker malware, several things can be noted. First, the System Flow Graph contains several processes. When excluding the Android processes like *m.android.phone* or *servicemanager*, two processes can be noted: *tor* and *libprivoxy.so*. It is uncommon to have more than one process for a benign Android application. Multiple processes reveal that the malware have launched another application or a native library. In particular, the graph shows a file *torrc* that is wrote by the process *org.SimpleLocker* and read by the *tor* process. Then *tor* connects to several IPs. We can easily suspect that this malware tries to communicate with the attacker using the Tor network.

Second, the Method Control Flow and Bytecode View gives an overview of the code. Two entry points (green) are identified: *onCreate*, the standard way of creating an Android Activity and *onStartCommand* which is used to start an Android Service. Nine methods have been identified as suspicious (red). One of the most interesting is *doShellCommands* which name is highly suspicious. Clicking this method shows the bytecode. The user can view how the code tries to run shell commands using *java.lang.Runtime*. All other suspicious methods can be inspected but we already know that they have been flagged as suspicious (high score) due to API calls such as encryption, telephony, etc. Other displayed intermediate nodes (blue) participate to the paths of calls to reach the suspicious nodes. Finally, clusters of nodes can be expanded: sink clusters give access to subcalls of the suspicious nodes. Expanding the parallel clusters shows nodes that are in other packages. For example, the package *org.spongycastle* which is a repackaged version of an encryption library (The Legion of the Bouncy Castle). Nevertheless, as these nodes are in the parallel cluster, it seems that the malware does no use this library, which is confirmed by the dynamic analysis.

[1] The visualization of SimpleLocker using GroDDViewer is available at: `http://kharon.gforge.inria.fr/dataset/SimpLocker_sample_fd694cf5ca1dd4967ad6e8c67241114c.html`

5.2 Dynamic analysis

Using the replay capability of GroDDViewer gives an insight about the malware actions. The Interaction frequency graph shows a lot of interactions on the interval $[0, 100]$. When replaying, this first part corresponds to Android routines and are not linked with the malware execution that starts later. SimpleLocker starts at time $t = 250$: after being unpacked from the *.apk* file, it deploys local files like *torrc* and *privoxy.config*. Then, a long interaction is observed at time $t = 258$ with a file ending by *.enc*. This means that some long operations are running for this file. At timestamp 960, operations are finished on this file. At the end of the replay, we also see some interations between the *tor* process and some IPs.

This first dynamic overview suggests to focus on the interval $t > 240$. Thus, the user can use the zoom functionality to put the replay window on $t \in [240, 1000]$.

Then, the dynamic replay of Method Control Flow graph shows a sequence of calls *onCreate* $\rightarrow$ *run* $\rightarrow$ *encrypt* at times t near 250. It corresponds to the generation of the encrypted file *.enc* after starting the main activity of the application. Indeed, if the user inspects the executed *encrypt* method, as shown in Figure 3, the first lines of the bytecode shows the code *$r1 = new org.SimpleLocker.AesCrypt* followed by *specialinvoke $r1.("jndlasf074hr")* which corresponds to the call to the constructor of the used AES encryption algorithm with a constant encryption key. When opening the other animated nodes such as *findExistingProc*, *findProcessIdWithPidOf*, the user may think that it corresponds to the control of the Tor process for handling communication, which is less interesting to investigate.

Thus, the replay shows the encryption occurring at time $t > 250$ with the AES algorithm with a constant key. Unfortunately, as the communication is handled by a native independent process, we cannot inspect using the Method Control Flow graph the details of the execution of the communication protocol.

6 Conclusion

In this paper, we have presented GroDDViewer, an online tool for analyzing, understanding and replaying Android malware. GroDDViewer presents a dual view of malware: the graph of interactions that represents the operations that occured at operating system level and the graph of the methods of the bytecode. The presented use case illustrates how the user can easily gain some knowledge on the execution of a malware. Of course, such a tool cannot replace a manual investigation of the details of the bytecode but ease the understanding of the malware behavior. Future works concern the evaluation of the tool on a large panel of Android malware. Security analysts that conduct regular analysis of new samples will be involved in a campaign with two groups: one using GroDDViewer and not the other. Such a study will help to evaluate finely the obtained benefits.

References

1. Abraham, A., Andriatsimandefitra, R., Brunelat, A., Lalande, J.F., Viet Triem Tong, V.: GroddDroid: a Gorilla for Triggering Malicious Behaviors. In: 10th International Conference on Malicious and Unwanted Software. pp. 119–127. IEEE Computer Society, Fajardo, Puerto Rico (oct 2015). https://doi.org/10.1109/MALWARE.2015.7413692
2. Andriatsimandefitra, R., Tong, V.V.T.: Capturing Android Malware Behaviour Using System Flow Graph. In: The 8th International Conference on Network and System Security. pp. 534–541. Springer Berlin / Heidelberg, Xi'an, China (oct 2014). https://doi.org/10.1007/978-3-319-11698-3_43
3. Bodden, E.: Harvesting Runtime Values in Android Applications that feature Anti-Analysis Techniques. In: Network and Distributed System Security Symposium. pp. 21–24. No. February (2016)
4. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: Behavior-Based Malware Detection System for Android. In: 1st ACM workshop on Security and privacy in smartphones and mobile devices. p. 15. ACM Press, Chicago, USA (oct 2011). https://doi.org/10.1145/2046614.2046619
5. Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M.S., Conti, M., Rajarajan, M.: Android Security: A Survey of Issues, Malware Penetration and Defenses. IEEE Communications Surveys & Tutorials **PP**(99), 1–27 (2015). https://doi.org/10.1109/COMST.2014.2386139
6. Fratantonio, Y., Bianchi, A., Robertson, W., Kirda, E., Kruegel, C., Vigna, G.: TriggerScope: Towards Detecting Logic Bombs in Android Applications. Ieee S&P pp. 1–33 (2016). https://doi.org/10.1109/SP.2016.30
7. Grégio, A.R.a., Santos, R.D.C.: Visualization techniques for malware behavior analysis. Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense X **8019**, 801905–801905–9 (2011). https://doi.org/10.1117/12.883441
8. Kiss, N., Lalande, J.F., Leslous, M., Viet Triem Tong, V.: Kharon dataset: Android malware under a microscope. In: The Learning from Authoritative Security Experiment Results Workshop. The USENIX Association, San Jose, United States (may 2016)
9. Li, L., Li, D., Bissyande, T.F., Klein, J., Le Traon, Y., Lo, D., Cavallaro, L.: Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. IEEE Transactions on Information Forensics and Security **12**(6), 1269–1284 (jun 2017). https://doi.org/10.1109/TIFS.2017.2656460
10. Lindorfer, M., Neugschwandtner, M.: ANDRUBIS-1,000,000 Apps Later: A View on Current Android Malware Behaviors. In: 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security. IEEE Computer Society, San Jose, CA, USA (sep 2014)
11. Long, A., Saxe, J., Gove, R.: Detecting Malware Samples with Similar Image Sets. The Eleventh Workshop on Visualization for Cyber Security pp. 88–95 (2014). https://doi.org/10.1145/2671491.2671500
12. Paturi, A., Cherukuri, M., Donahue, J., Mukkamala, S.: Mobile malware visual analytics and similarities of Attack Toolkits (Malware gene analysis). In: 2013 International Conference on Collaboration Technologies and Systems (CTS). pp. 149–154. IEEE (may 2013)
13. Quist, D.A., Liebrock, L.M.: Visualizing compiled executables for malware analysis. In: 2009 6th International Workshop on Visualization for Cyber Security. pp. 27–32. IEEE (2009)

14. Tam, K., Khan, S., Fattori, A., Cavallaro, L.: CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In: 22nd Annual Network and Distributed System Security Symposium. San Diego, California, USA (feb 2015)
15. Trinius, P., Holz, T., Gobel, J., Freiling, F.C.: Visual analysis of malware behavior using treemaps and thread graphs. In: 2009 6th International Workshop on Visualization for Cyber Security. pp. 33–38. IEEE (2009). https://doi.org/10.1109/VIZSEC.2009.5375540
16. Wagner, M., Fischer, F., Luh, R., Haberson, A., Rind, A., Keim, D.A., Aigner, W.: A Survey of Visualization Systems for Malware Analysis. EuroVis (2015). https://doi.org/10.2312/eurovisstar.20151114
17. Wagner, M., Aigner, W., Rind, A., Dornhackl, H., Kadletz, K., Luh, R., Tavolato, P.: Problem Characterization and Abstraction for Visual Analytics in Behavior-based Malware Pattern Analysis. Proceedings of the Eleventh Workshop on Visualization for Cyber Security pp. 9–16 (2014). https://doi.org/10.1145/2671491.2671498
18. Weichselbaum, L.: Andrubis: Android Malware Under The Magnifying Glass. Tech. rep. (2014)
19. Yan, L.K., Yin, H.: DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In: USENIX Security Symposium. p. 29. USENIX Association (aug 2012)
20. Zhuo, W., Nadjin, Y.: MalwareVis: Entity-based Visualization of Malware Network Traces. In: The Ninth International Symposium on Visualization for Cyber Security. pp. 41–47. ACM Press, New York, New York, USA (2012). https://doi.org/10.1145/2379690.2379696