Visual Analysis of Malware Behavior Using Treemaps and Thread Graphs

Philipp Trinius* Thorsten Holz† Jan Göbel‡ Felix C. Freiling§

Laboratory for Dependable Distributed Systems, University of Mannheim, Germany

ABSTRACT

We study techniques to visualize the behavior of malicious software (*malware*). Our aim is to help human analysts to quickly assess and classify the nature of a new malware sample. Our techniques are based on a parametrized abstraction of detailed behavioral reports automatically generated by sandbox environments. We then explore two visualization techniques: *treemaps* and *thread graphs*. We argue that both techniques can effectively support a human analyst (a) in detecting maliciousness of software, and (b) in classifying malicious behavior.

Keywords: Invasive Software, Information Visualization, Behavior Analysis

1 Introduction

The field of malicious software (*malware*), to which worms, bots and other unpleasant artifacts belong, is one of the most active and also one of the most threatening areas of computer security. In recent years, we are observing a huge increase in the number of malware samples collected by anti-virus vendors [10]. Therefore, it is mandatory that we develop tools and techniques to analyze new malware samples with no (or only limited) human interaction.

Analyzing malware is a non-trivial task since attackers use code obfuscation techniques like binary packers, encryption, or self-modifying code to evade analysis. The approaches that have been developed to analyze a given sample can typically be classified to perform either *static* or *dynamic* analysis. In static analysis the *code* of the sample is examined, for example by disassembling or decompiling the binary file. The main advantage of this approach is that we can obtain a complete overview of what a given software does. However, static analysis is usually cumbersome and time consuming since many techniques to evade static analysis have been developed. For example, an attacker can use self-modifying code, obfuscate the code [3], or use different techniques to prohibit static disassembly [7]. In general, static analysis has therefore many limitations [5]. Furthermore, static analysis does not allow a high degree of automation during analysis.

An alternative to static analysis is *dynamic analysis*. In dynamic malware analysis, the *behavior* of malware is analyzed, for example, by executing it within a debugger. One of the most promising approaches to the dynamic analysis of malware consists of *sandbox* solutions of which several ones have been developed in recent years [1, 6, 11]. A sandbox executes a malware sample in a controlled environment and records all system-level behavior such as modifications of the filesystem or the registry. As a result, the sandbox generates an analysis report summarizing the observed behavior of the sample. In contrast to static analysis, sandbox-based dynamic analysis can be automated to a high degree.

*contact author, e-mail: trinius@informatik.uni-mannheim.de
†e-mail: holz@informatik.uni-mannheim.de
‡e-mail: goebel@informatik.uni-mannheim.de
§e-mail: freiling@informatik.uni-mannheim.de

6th International Workshop on Visualization for Cyber Security 2009
11 October, Atlantic City, New Jersey, USA
978-1-4244-5415-0/09/$25.00 ©2009 IEEE

Sandbox reports, often consisting of several hundred entries, usually offer a level of detail that allows careful analysis but overwhelm a human analysts who is interested only in a quick assessment of a particular sample. Given the fact that many samples are just small variants of a well-known malware family, there is a clear need to make the behavior of malware samples more and easier accessible to a human analyst. In this paper, we explore a combination of abstraction and visualization to do exactly this.

Contributions. We first present a parametrized method to abstract sandbox reports into increasingly simple summaries. Then we use classical techniques to visualize the resulting reports and make them accessible to the human reader. We argue that the combination of both techniques offers unique insights into the main features of malware such that humans are effectively supported in detection malicious behavior when it occurs. Furthermore, we argue that the visual representation allows an analyst to rapidly classify the behavior of new malware samples if it belongs to one of the large and well-known families of malware families currently spreading in the wild.

We use two visualization techniques: treemapping and thread graphs. *Treemapping* [8, 9] displays the distribution of the individual operations performed by a sample. The resulting treemap presents this information as a set of nested rectangles and provides a quick overview of the main overall behavior of the sample, e.g., whether the main task of the sample lies in the area of network interaction, changes to the file system, or interaction with other processes. Since tree maps display nothing about the sequence of operations, we use *thread graphs* to visualize the temporal behavior of the individual threads of a sample. A thread graph can be regarded as a behavioral fingerprint of the sample. An analyst can then study this behavior graph to quickly learn more about the actions of each individual thread.

We demonstrate the practical feasibility of our approach by performing several case studies. First, by analyzing four different malware classes, we show that our methods can actually be used to find samples belonging to the same family of malware. Second, we show how our techniques can also be used to *detect* malicious activity by analyzing several malicious data files as for example PDF files that exploit a vulnerability in Acrobat Reader.

Related Work. The work by Xia et al. [12] on visual analysis of program flow data shares with our work the goal to visualize the *behavior* of a given malware sample. While Xia et al. focus on data propagation and taint tracking, we use a simple abstraction technique to visually summarize the observed behavior for a human analyst. Our thread graphs are similar to the visualization of Xia et al., but contain more information on the actual behavior of each thread. We are not aware of any work which has used treemapping to visualize the behavioral information.

Different techniques to support reverse engineering of binary and data files with visualization were introduced by Conti et al. [2]. Similar to our system, the goal is to support a human analyst, however Conti et al. use static analysis techniques. Since malware is commonly packed using an executable packer, the distribution of the individual bytes is very similar and structural information is lost. When dealing with malware, the techniques of *byteview* and *byte presence* visualization of Conti et al. cannot be easily applied.

Roadmap. This paper is structured as follows: Section 2 provides an overview of system. Section 3 describes our approach to visualize the behavior of a malicious sample based on parametrized abstractions. We discuss a detailed malware visualization example in Section 4, present the technique of visual malware clustering in Section 5 and show how non-executable files can be visualized in Section 6. We conclude the paper in Section 7 with a summary and ideas for future work.

2 System Design

Figure 1 provides a schematic overview of our system. We use a large database of malware samples collected using honeypots. To analyze these samples, we first execute them in a controlled, instrumented environment (*sandbox*) and observe their behavior during runtime. In the second step, we visualize the collected information using different techniques. As an output, our technique generates different visualizations of the sample's behavior, which enable a human analyst to get a quick overview of what a given malware sample does.

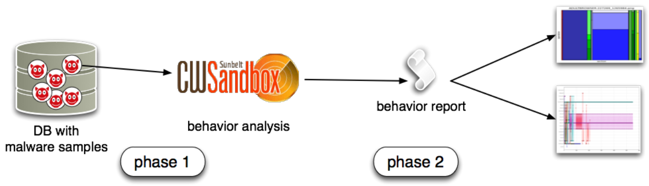

Figure 1: Overview of malware behavior visualization system

The advantage of dynamic analysis is it partly circumvents the problems of code obfuscation techniques like packers or crypters. Since the software knows how to unpack itself, we can simply execute it, let the sample unpack/decrypt, and then we can study its behavior. Furthermore, this approach can be automated to a high degree. In our case, we opted for *sandbox* solutions to perform dynamic analysis. During execution, the sandbox observes the system-level changes at runtime, e.g., changes to the filesystem or the Windows registry, or network packets sent and received. The resulting output of a sandbox is a behavior report which summarizes the observed activity.

In our system we use behavior-based malware analysis based on CWSandbox [11]. We execute the sample we want to analyze for two minutes in the CWSandbox environment and let the tool record all system-level activity. The result of the analysis phase is a report in XML format containing information about the observed behavior for each thread: it includes for example information about all loaded system libraries, outgoing and incoming network connections, accessed or manipulated registry keys, and many more events. With our CWSandbox installation we generate between 2,500 and 4,000 such reports on a typical day. We thus obtain a large number of fine-granular reports detailing the actions of each thread executed by the sample.

Note that behavior-based analysis enables us to analyze both executable *and* data files. Data files (like word processor documents) can be "indirectly" analyzed by opening them in its associated application and then observing the behavior of the application. For example, we can analyze a given PDF file by opening it with Acrobat Reader and then analyzing the behavior of this program. We show later that this can be used to detect data files behaving in an anomalous way.

3 Abstraction-based Visualization

Each sandbox report is usually rather long (i.e., typically several thousand bytes in XML format) since CWSandbox analyzes many API functions and logs all associated activity. For an effective visualization it is necessary to condense or abstract information in these reports. Therefore we transform a given XML report generated by CWSandbox to a shorter and better display format: we map each API call to a specific *section* that groups API calls with similar functionality together. For example, all API calls related to file system activity belong to one section. Furthermore, we order the arguments of each API call according to their relevance, i.e., more significant arguments are placed first and less significant arguments are completely omitted.

3.1 Abstraction Levels

Using this idea, there are four different and novel abstraction levels at which sandbox reports can be generated:

- To obtain a coarse overview of what a process is doing, we just display the individual sections of a process, e.g., the mere fact whether a sample accessed the registry or not. This is termed *level 1* abstraction.
- *Level 2* abstraction extends level 1 by also showing the names of the API calls executed in each section.
- *Level 3* abstraction extends level 2 by adding also information about the most significant argument of each API call.
- *Level 4* abstraction takes even more arguments into the display format, adding further details.

We show below that especially level 2 and 3 abstraction are useful when visualizing malware behavior.

3.2 Visualization

We implemented two different approaches for visualizing the behavior reports which both help a human analyst to quickly understand which actions a given sample performs: a *treemap* provides a summarized overview of the actions performed by a malware sample and their frequency, while a *thread graph* visualizes the behavior of the individual threads of a process.

3.2.1 Treemap

In the first visualization approach, we use the technique of treemapping to transform the behavior report into a standardized format. A treemap displays information as a set of nested rectangles and we need to define a tiling algorithm to specify the construction of the treemap: The width of each rectangle is proportional to the percentage of API calls from the behavior report belonging to this section, i.e., if more API calls belong to a specific section, then the according rectangle is wider. We also split each rectangle according to the operation of this API call to obtain another dimension within the treemap. Overall we observe over 120 distinct API calls out of 20 sections. Within the treemaps the individual sections are plotted in a fix order and color. The leftmost red colored section for instance is the so called `com section` which consists of three API calls. Next is the `dll handling section` which is plotted in blue color tones. Since some reports do not posses API calls of all sections, the corresponding treemaps do not show all sections as well. To interpret the information that is encoded within the treemaps, both the position and the color has to be considered. An example treemap of a malware labeled as *Adultbrowser* is shown in Figure 2. We can see that two sections are significantly wider than the remaining sections, indicating that this sample performs some specific operations rather frequently. The height of the rectangles for a given section also visualizes the frequency of an operation within this section.

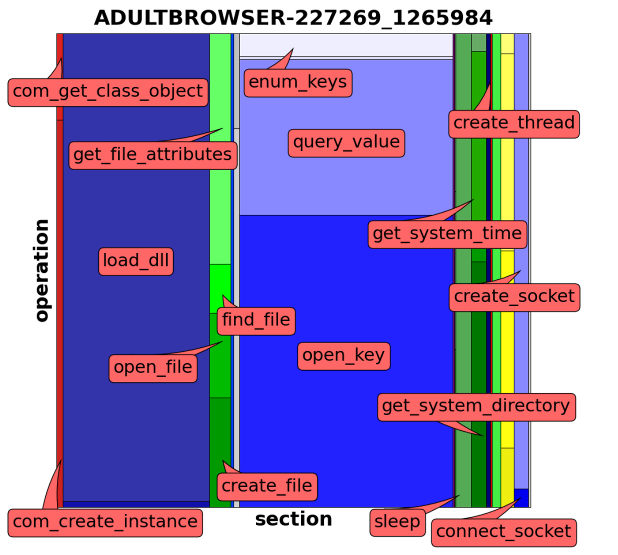

Figure 2: Graphical representation of *Adultbrowser* malware using a treemap

3.2.2 Thread Graph

Our second approach is the visualization of the actual chronological behavior of a given sample. We generate a graph representing the temporal order of executed system commands and the different threads spawned by a binary. The x-axis represents the time (sequence of performed actions), while the y-axis indicates the operation/section of the performed action. Based on this construction, we called this visualization a *thread graph*.

Since a thread may consist of thousands of operations, a threshold is necessary to achieve a clear representation and to have a consistent visualization. Therefore, a thread graph only shows a certain maximum number of operations per thread. In our experiments a threshold of 550 operations has proven to be a good tradeoff between accuracy and clarity.

Figure 3 shows the same *Adultbrowser* malware sample as displayed previously, this time using the thread graph representation. We see that one thread is responsible for the majority of operations for this sample. This thread performs many registry operations and initially performs many network- and system-related operations (operations 90-140). Additionally, two more threads are spawned, but they perform only a limited amount of operations during the analysis phase.

These thread graphs, depending on the duration of the execution time of the malware sample in a sandbox, represent the unique behavior for a certain family of malware. This means that any other sample that generates the exact same (or at least a very closely related) graphical representation of its behavior can be considered to be related to the previously determined type of malware. We show examples for this relationship in the next section based on several case studies.

4 EXAMPLE VISUALIZATION OF HOOKSHELL

We now discuss a single selected malware-sample and its visualization in detail. The file with the MD5 hash a9f6aa1649e6a0f1bfad8a576f0193a0 was analyzed April 29, 2009 and was classified as `Hookshell` malware by several anti-virus engines, e.g., Antivir, SecureWeb-Gateway and Sunbelt. The CWSandbox report is quite small and consists of only 379 lines describing the malware's behavior.

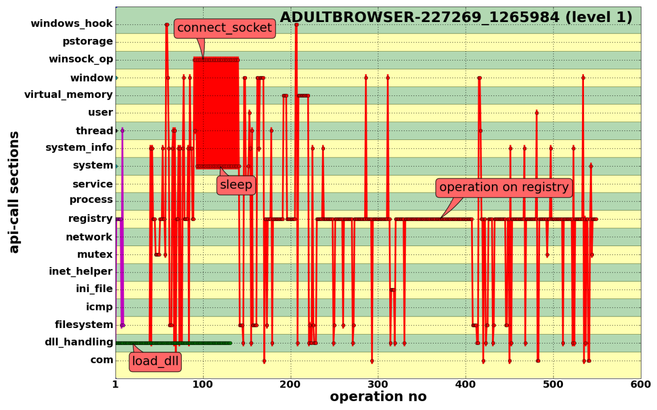

Figure 3: Graphical representation of *Adultbrowser* malware using a thread graph

Figure 4 shows the treemap generated from the CWSandbox report. The image clearly shows that the malware executes API calls from six different sections. From left to right these are the `dll handling section`, the `filesystem section`, the `ini file section`, the `registry section`, the `process section`, and the `system info section`. The horizontal subdivision of all sections show that at least two different operations per section were executed.

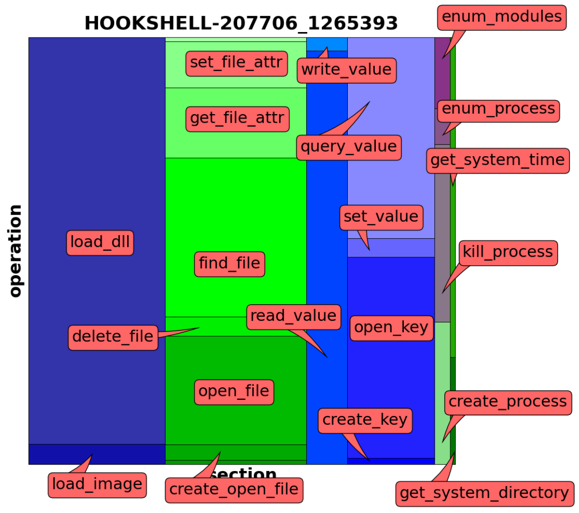

Figure 4: Treemap of one Hookshell malware sample

For a detailed analysis of the malware behavior we have to consider the thread graphs displayed in Figures 5 and 6. The level 1 thread graph shows that the malware executes operations from six sections. Each thread can be identified together with the operations it performs. From the thread graph it is possible to determine four distinct threads running different API calls. Two threads execute exactly the same sequence of API calls, therefore their representation color turned black [1]. To circumvent this overlay of two or more threads, the graph could be split up into one graph per thread.

By taking a closer look at the single threads (level 2, displayed in Figure 6), we can still extract detailed information about the malware's operations. All threads initially load a set of libraries, explained by the straight black line at the beginning. The red colored

[1]The different colors sum up to black.

thread investigates the filesystem, changes some file attributes and creates a new file (operation number 29). Afterwards it reads some registry keys and frequently queries an ini file before it finally creates a new registry key. These operations seem to be the key feature of the malware binary, as the other threads do not perform any operations on the filesystem. The exact files and registry keys that were read and created cannot be determined from the thread graphs, but need to be extracted from the detailed sandbox report.

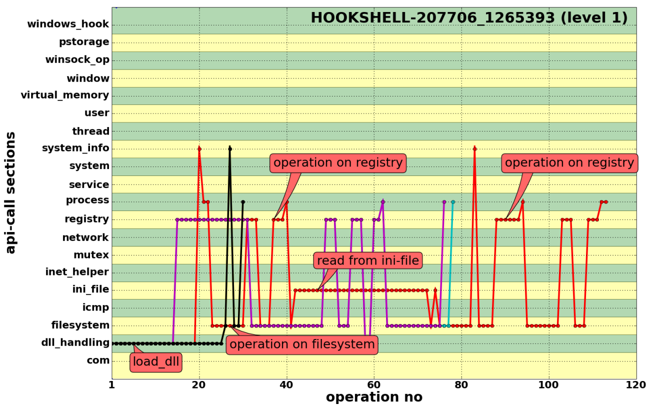

Figure 5: Level 1 thread report of one Hookshell malware sample

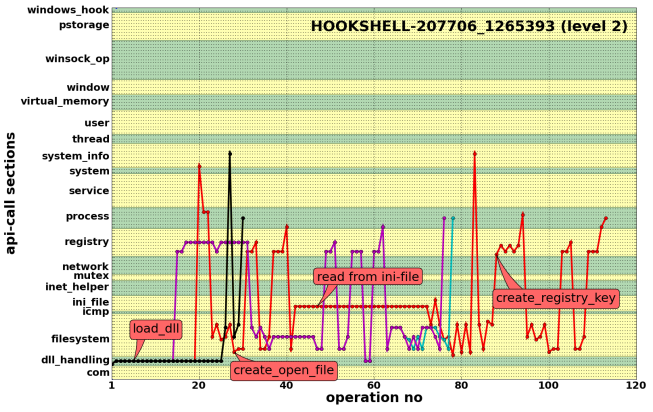

Figure 6: Level 2 thread report of one Hookshell malware sample

5 Visual Malware Clustering

We now show how the visualized malware behavior reports can be used to quickly determine if a given sample belongs to a certain malware family. This kind of clustering is a good way for malware analysts to determine if a new sample needs to be examined in more depth or if it belongs to an already know class of malware.

As a test set we used a sample set of 2,000 malware samples that were pre-classified and labeled using a set of six anti-virus softwares. Thus, every image contains the according label in its filename. Altogether, we have 13 different labels, namely `Adultbrowser`, `Allaple`, `Bagle`, `Bancos`, `Hookshell`, `Looper`, `Podnuha`, `Porndialer`, `Rotator`, `Sality`, `Spybanker`, `Sramler` and `Swizzor`.

Therefore, in the best case an analysis would divide the treemaps into 13 distinct groups, containing only the according images. For most of the examined malware samples this is in fact possible. We analyzed the malware samples with CWSandbox and visualized the observed behavior with our approach. Figure 7 shows eight treemaps out of four differently labeled classes and substantiates the idea of using the treemaps as input for an image-based clustering. The images of different malware samples are visually different, but samples of the same malware are almost identical. Nevertheless, this does not apply for all classes. Since the results of behavior-based analysis of malware depends on several variables which are beyond the analyst's control (e.g., in case the command and control server of a bot is unreachable, the bot will show an entirely different behavior) a perfect clustering seems unfeasible.

6 Visualization of Non-Executable Files

Our visualization approach is not limited to executable files only. Opening a data file with its associated application within a sandbox environment allows us to generate a behavior report for almost any data format. As this report is the basis to our visualization process we are able to visualize the behavior of a PDF file, for example, by running the appropriate PDF reader together with the document we want to analyze in a sandbox. In this section we briefly explain the idea of indirect behavior analysis and visualization with the help of benign and malicious PDF documents.

As a data set we used 17 malicious and 200 randomly chosen benign PDF files extracted from a Google search for PDF documents. We scanned each of these files with the anti-virus engine of Avira. All 200 PDF documents were marked as not infected. For our experiments we used 17 PDF documents that Antivir recognized as being malicious. 15 of these files show anomalous behavior in our sandbox environment, as these files open a network connection during the analysis phase. We use the english version of Adobe Acrobat Reader 8.0 for viewing the PDF files and manually confirmed that these files are indeed malicious.

Depending on the instrumented environment and the checks malware authors may have built into their software, the malicious code of an executable or data file may be executed or not. Related to our example, the malware author may check whether or not a vulnerable version of Adobe Acrobat Reader is installed.

As a result of the indirect behavior-based analysis we observed that the Acrobat Reader does not show the same behavior for all benign data files. Based on their visualized behavior the benign PDF files can be divided into two classes. Figure 8 (a) and (b) show examples of treemaps belonging to one of the two classes. Both classes can be distinguished by their leftmost operation, colored red in image (a) and are missing in image (b). This additional operations are API calls of the so called `com section`. This distinctive feature can also be observed in the thread graphs of the PDF files, shown in Figure 9 in the upper row.

The reason for executing calls from this section depends on the defined objects within a PDF file and are no criteria for malicious or suspicious behavior.

For malicious PDF files the behavior-based reports and the resulting visualizations are not that homogeneous. This is due to the fact that the behavior report also contains the operations of additional processes initiated by the analyzed malicious files. The extra processes performing different operations belong to binaries downloaded by injected commands in the malicious PDF documents. Figure 8 (c) and (d) show treemaps of two malicious PDF files. Even though, at a first glance, the two treemaps look very similar to the treemap (b) of the same figure, there are three obvious differences that distinguish treemaps of malicious PDF documents from those of benign PDF files:

1. The violet/blue colored section of the malicious PDF files is slightly expended.

2. The treemaps of the malicious PDF files have an additional, light-green colored section to the left of the yellow colored section.

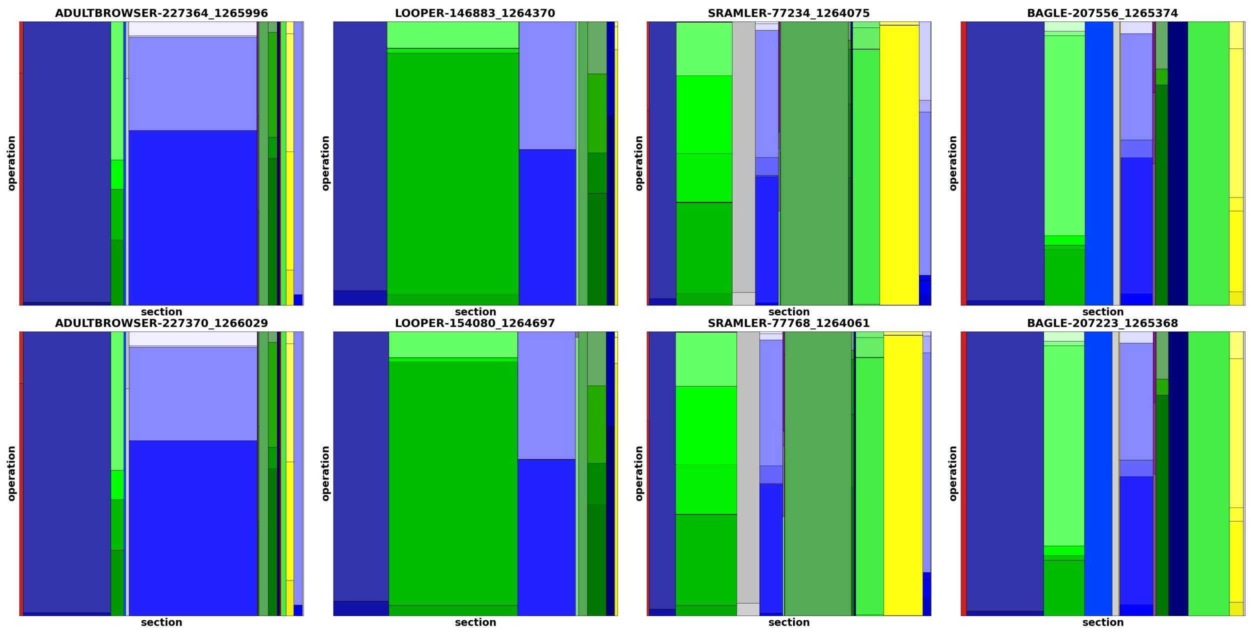

Figure 7: Selection of treemaps out of four different malware classes

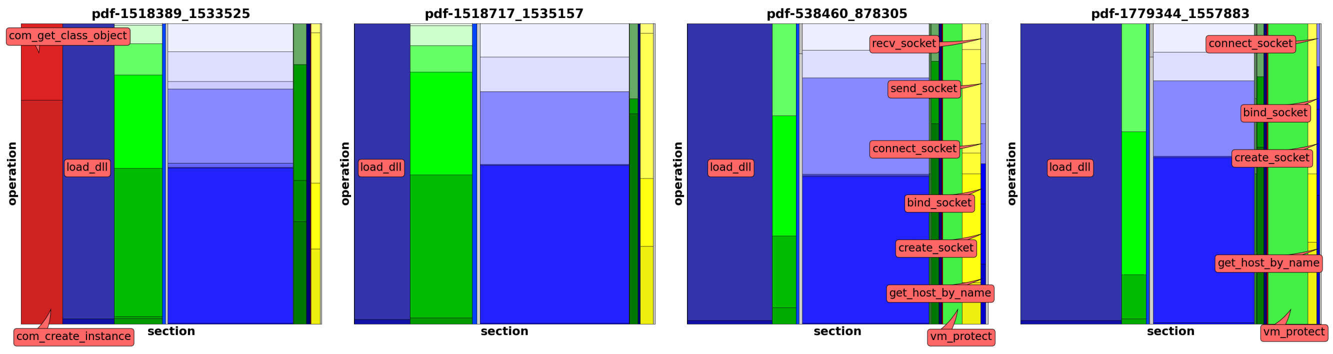

Figure 8: (a)-(b) treemaps of benign PDF files. (c)-(d) treemaps of malicious PDF files.

3. Only the treemaps of the malicious PDF documents have a mentionable wide section on the right of the yellow colored section.

As a result we are able to distinguish between benign and malicious PDF documents just by comparing the corresponding treemaps. In the following we will describe the API calls which are responsible for these differences and show why it is possible to use them as a basis for decision-making.

The violet/blue colored section in the treemaps corresponds to the `dll handling section`. This section contains the three API calls `load_image` `load_dll` and `get_proc_address`. Dynamic Link Libraries (DLLs) are files that provide helpful functions to a calling program. As every Microsoft Windows based software loads a few of those libraries at every startup, this section is shown in every treemap. The size of this section is determined by the number of DLLs loaded. This behavior is also visible in the thread graphs, as shown in Figure 9. With benign PDF thread graphs there is a single red colored line indicating the loading of DLL files. The thread graph of the malicious files shows a second green colored line – another thread – loading significantly more libraries. The loading of several more libraries can be used as a basis for decision-making, as the number of needed DLLs depends on the application only and not on the document that is opened.

Next we take a look at the light-green colored section to the left of the yellow section. This section corresponds to the so called `virtual memory section` of a CWSandbox report which contains all API calls needed to allocate, read and write to virtual memory. The malicious PDF documents need this additional memory to save variables and code, generally exploit-related code. As benign PDF files run within the Acrobat Reader, they do not need this kind of memory allocation.

The third discriminative feature is the additional section to the right of the yellow colored section. Although this part is hard to spot, this additional section is the most important in distinguishing a benign PDF from a malicious one. It is the so called `winsock operation section` that contains all network related API calls. Considering the thread graphs of Figure 9 (c) and (d) there are two additional threads – the violet and the yellow colored ones – which execute operations belonging to this section. These are the operations to download malware to the exploited system. A PDF document that initiates network connections to download additional software can truly be identified as being malicious.

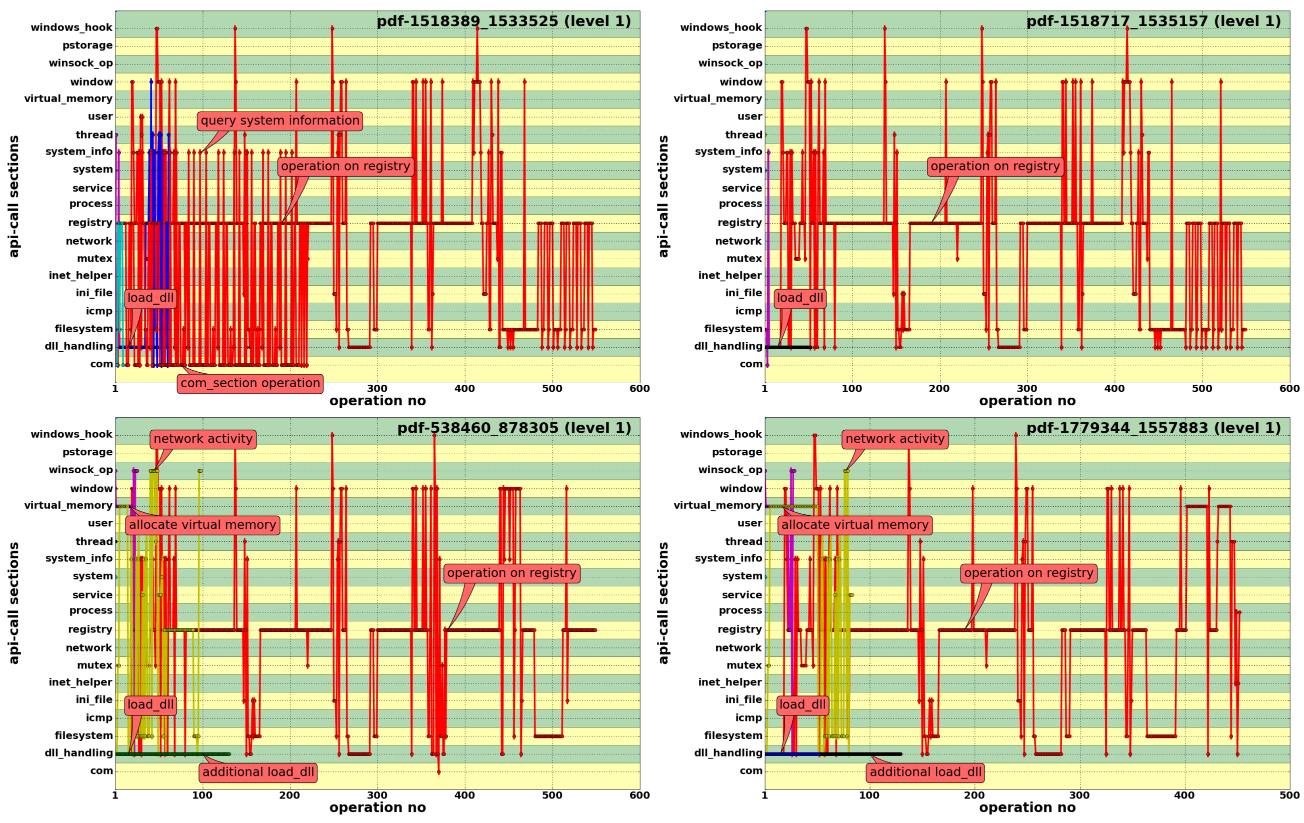

Figure 9: Upper row (a)-(b): Level1 thread graphs of two benign pdf. Lower row (c)-(d): Level1 thread graphs of two malicious pdf.

7 Conclusion and Future Work

In this paper, we presented two abstraction-based visualizations of malware behavior. In our work on malware analysis we have found these visualizations very helpful. We are interested in getting feedback of other analysts to enhance the representation. Although we extensively use CWSandbox, our visualization approach is independent of a particular sandbox solution.

The problem with the behavior-based approach is that we only see one execution path of a binary and thus may miss important functionality during the analysis process. With multipath execution analysis [4], this limitation can be overcome to a certain extend. Nevertheless, this drawback of dynamic analysis has to be considered. Another drawback of behavior-based analysis is the fact that the analyzed binary can detect the presence of the analysis environment and then behave differently.

Possible extensions of our work could be some kind of zoom-functionality, which allows an analyst to zoom in and out of a treemap. Starting with global treemaps, like the ones we present in this paper, the information could be enriched with every zoom factor. Thus an analyst could investigate unusually looking parts of the analysis report in detail.

An interesting line of future work is to study existing image clustering and classification algorithms and compare the results of these algorithms on our database with the labeling of anti virus vendors or clusterings which are based on the full behavior reports. Furthermore, since the visual classification is much faster than a classification based on the detailed reports visual classification could be used as preprocessing or even prediction for a precise classification.

Acknowledgments. This work has been supported by the Bundesamt für Sicherheit in der Informationstechnik (BSI).

References

[1] U. Bayer. Anubis: Analyzing Unknown Binaries. http://analysis.seclab.tuwien.ac.at/.

[2] Gregory Conti, Erik Dean, Matthew Sinda, and Benjamin Sangster. Visual Reverse Engineering of Binary and Data Files. In *Workshop on Visualization for Cyber Security (VizSec)*, 2008.

[3] Cullen Linn and Saumya Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *ACM Conference on Computer and Communications Security (CCS)*, 2003.

[4] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy*, 2007.

[5] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of Static Analysis for Malware Detection. In *Annual Computer Security Applications Conference (ACSAC)*, 2007.

[6] Norman ASA. Norman SandBox. http://sandbox.norman.no/.

[7] Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. Binary obfuscation using signals. In *USENIX Security Symposium*, 2007.

[8] Ben Shneiderman. Treemaps for space-constrained visualization of hierarchies. Internet: http://www.cs.umd.edu/hcil/treemap-history/.

[9] Ben Shneiderman. Tree Visualization With Tree-Maps: 2-D Space-Filling Approach. *ACM Trans. Graph.*, 11(1), 1992.

[10] Symantec. Internet Security Threat Report Volume XIV. Internet: http://www.symantec.com/business/theme.jsp?themeid=threatreport, April 2009.

[11] Carsten Willems, Thorsten Holz, and Felix Freiling. CWSandbox: Towards automated dynamic binary analysis. *IEEE Security and Privacy*, 5(2), March 2007.

[12] Ying Xia, Kevin Fairbanks, and Henry Owen. Visual Analysis of Program Flow Data with Data Propagation. In *Workshop on Visualization for Cyber Security (VizSec)*, 2008.