Visualizing Compiled Executables for Malware Analysis

Daniel A. Quist*
New Mexico Tech
Los Alamos National Laboratory

Lorie M. Liebrock†
New Mexico Tech

ABSTRACT

Reverse engineering compiled executables is a task with a steep learning curve. It is complicated by the task of translating assembly into a series of abstractions that represent the overall flow of a program. Most of the steps involve finding interesting areas of an executable and determining their overall functionality. This paper presents a method using dynamic analysis of program execution to visually represent the overall flow of a program. We use the Ether hypervisor framework to covertly monitor a program. The data is processed and presented for the reverse engineer. Using this method the amount of time needed to extract key features of an executable is greatly reduced, improving productivity. A preliminary user study indicates that the tool is useful for both new and experienced users.

Keywords: Reverse Engineering, Visualization, Dynamic Analysis

Index Terms: K.6.1 [Management of Computing and Information Systems]: Project and People Management—Life Cycle; K.7.m [The Computing Profession]: Miscellaneous—Ethics

1 INTRODUCTION

Modern day compiled executables are extremely large and complicated. Furthermore the language available for reverse engineering is assembly and it is a daunting task to analyze programs. The learning curve necessary to master reversing is quite steep. Once the skills are mastered, the process is inherently labor intensive and therefore costly.

The primary task of a reverse engineer is to determine the functionality of a program. Although determining the intent of the code would be valuable, it is even more difficult to discern than the functionality is. Reverse engineers use a variety of tools to assist with the process. These include static disassemblers, debuggers, system call trackers, and scripting tools. Tracking the execution of a program is a task that is difficult as malware actively tries to avoid detection. It is imperative that modern tools measure and analyze executables with no detectable impact to the executable.

Most often when analyzing a particular executable, knowing where to start is the biggest problem. The amount of information presented can be overwhelming. Experience and skill can aid in this process. Reducing the amount of time necessary to understand the overall program layout yields large increases in reverse engineer productivity. When a program is obfuscated by packers and other software armoring tools, this further slows down the analysis. With this in mind we present our framework to facilitate initial program comprehension and deobfuscation.

Two main classes of tools exist: static and dynamic tools. Static analysis tools include disassemblers (such as IDA Pro), string searching tools, and signature matching systems (including many anti-virus programs). Dynamic analysis tools include system call and operating system state tracking, such as the Sysinternals' Processmon tool, and debuggers, such as OllyDbg, SoftICE, and GDB. The run-time of an executable is closely controlled with these tools.

To aid in the reverse engineering process we set out with the following goals. First, quickly determine the original entry point of packed or compressed executables. This is a large task that can be complicated depending upon the sophistication of the program. Second, aid in the understanding of the overall composition of the program. Specifically highlight the most commonly executed portions of the program.

This paper makes the following contributions: We show that the overall process of analyzing malware and other executables can be shortened via our visualization tool. We provide a covert method for monitoring running programs via modifications to the Ether framework. Finally, we integrate all of our tools with established reverse engineering tools to speed analysis.

The paper is organized in the following sections. A discussion of related work is in Section 2. Section 3 outlines a reverse engineering process. In section 4, we outline the VERA (Visualization of Executables for Reversing and Analysis) architecture. This includes modifications to Ether, data organization and graph layout, and finally the presentation system. Next, in section 6, we apply this visualization to the Mebroot worm. We then show results from our user study in section 7. Finally, we end with the conclusion and future work section.

2 PREVIOUS AND RELATED WORK

Using hardware virtual machine hypervisors for monitoring program execution was discussed extensively by Dinaburg et al. [5] for the use of malware analysis. The discussions of the modifications we have made to this framework are discussed in section 4.1. Royal used a similar system but with the Linux kernel virtual machine architecture [16]. Using Dynamic instrumentation systems to effectively monitor program execution was suggested by the SPiKE framework [20]. Run-time debugging of malware has also been proposed by Cifuentes et al. as a method for rapidly understanding program execution [4]. The PIN system has been used by Ma et al. for tracing the injection of malicious code into a vulnerable service [13]. The importance of analyzing dynamic behavior was also illustrated by the TTanalyze tool [2] [14]. Modifications to traditional virtual machine architectures have been used to aid in malware analysis in [12, 17].

Visualization of program execution has been used in the past with good results. Analysis has centered on programs with source code available, or for security analysis of unobfuscated code (such as Microsoft executables). Xia et al. monitored system calls of a running executable to show taint propagation and system call flow for a process [21]. Other systems such as those presented by Telea and Voinea demonstrate the effectiveness with available source code [19]. Bohnet et al. provide a method to visually explore C and C++ source code [3]. Similar to our technique, they emphasize the importance of distilling a large program ($>$ 1 million lines of code) into its base portions. This distillation process is important to develop a high-level overview of the overall flow of a program to highlight relevant portions. The VERA architecture discerns itself from this work by focusing on compiled code with a focus on basic-block malware analysis.

*e-mail: dquist@nmt.edu
†e-mail: liebrock@cs.nmt.edu

Two commercial products provide static analysis of compiled executables. BinNavi by Zynamics provides a graph based analysis method for highlighting dependencies of a program [23]. This program highlights program flow and structure by using function calls. Responder by HBGary, Inc. likewise focuses on the function calls of software to highlight memory access, variables, and system calls [10]. These two products both rely on debugger interfaces or static analysis tools, which have been shown to be unreliable [6, 8, 7].

3 Reverse Engineering Work-flow

Reverse engineering is a process that can be very time consuming. Determining the function and intent of a program is difficult and requires a lot of patience. The analyst can very quickly develop fatigue from analyzing the code. The following process serves as a high-level technique for reverse engineering an unknown binary.

In this section we will define a sample method that we have found useful when reverse engineering a program. This is by no means the standard method, as no such thing exists, but one that we have found effective for reverse engineering.

The process can be broken down into the following steps: First set up an isolated environment. Next execute the program to look for any discerning output. Use tools to monitor changes to the operating system while executing. Third, load the program into tools such as IDA Pro to begin the reverse engineering process, deobfuscating the binary if required. Finally identify and analyze relevant and interesting portions to focus on.

While there may be many means to developing a sense of functionality and intent of a program, we have found this one to be very useful.

3.1 Setup of an Isolated Analysis Environment

Most reverse engineering deals with malicious or potentially malicious code which is the primary focus of this paper. It is important to have an isolated environment to contain any nefarious activity that might occur. The common practice is to use a virtualization system such as VMWare or Virtual PC. An operating system, most commonly Windows XP, is installed and configured in a manner that represents a common user's environment.

Each virtualization system should have the ability to take a snapshot of the current state of the system. Before any analysis is performed, this is a necessary baseline. First, it provides a known-good system to compare with subsequent system state during execution. Second, restoring to this configuration is often necessary to understand the full execution process of a program and allows for quick recovery to the pre-infection state. Once the snapshot is taken, the program can be executed and analyzed.

3.2 Execution and Initial Analysis

Executing the malware in a controlled virtualized environment provides safety to the analyst from infection. This type of reverse engineering is often referred to as dynamic reverse engineering. The goal is to capture the overall impact that the software has on a system without focusing on the program's actual code.

This gives us a high-level overview of what the program is doing. Looking at the changes affected on the operating system shows any modifications or destructive activities performed. Common tools such as Microsoft's Sysinternals tools monitor for system call executions, modifications to files, and registry modifications. Using tools such as Wireshark, the program's network activity can be monitored.

The goal of this section of analysis is to quickly determine what to look for inside of the disassembled code. Without understanding the initial behavior, discerning the meaning of the disassembly is substantially more difficult.

3.3 Deobfuscation

Many new malware samples are armored. This is done primarily to prevent signature detection, debugging, and reverse engineering. Removing these obfuscations is a necessary step prior analysis. These can be any manner of operations. Typical examples are encoding or encrypting the executable, detecting virtual machines, and detecting debuggers. Since code is the primary tool for understanding program execution, any protections must be removed. Guo et al. provide a good overview of the packing and compression problems in modern executables [8].

Solutions to removing packers can rely on static methods such as those by Guo or by using dynamic methods using virtual machines [5, 17, 16, 12], and page-fault assisted debugging [15, 18]. These systems rely on being able to monitor execution of a running process and determine where the deobfuscated or modified code is running. When this is done, a memory dump of the application's code is performed so that disassembly may occur.

3.4 Disassembly

Disassembling the binary involves loading it into common tools such as IDA Pro [11] or OllyDbg [22]. This is where the actual code comprehension occurs. IDA Pro provides an excellent interface to annotate and understand the code. The graphing methods are primitive but can help to explain the code on a very high level. The analysis of code can be broken into two classes: system call analysis and code comprehension.

System call analysis is the process of looking at the relevant library calls that are made by a portion of code. For instance if the application reads a file, then performs a network operation, and later closes a file, one can reasonably infer that this program is performing a file transfer. This is a very high-level analysis technique that provides a good high-level overview of the process. It is inherently static based analysis and can be easily subverted by software armoring systems. What it does not do is provide information about the actual code executed by the application.

Code comprehension is the process of discovering the algorithms or underlying structure of the program. This is typically done when an interesting portion of code is identified and needs to be understood more fully. For example, this analysis can be based on what was inferred with system call analysis. Specific portions of code that need to be identified include encryption or authentication algorithms and any obfuscation code. This identification process can be labor intensive.

3.5 Identify and analyze relevant portions

There is no set method for determining what is relevant and interesting in the reverse engineering process. While there are a few techniques, most often this is the step that beginners have the most trouble with. Some of the techniques that are successfully used are to look at interesting strings, look at relevant API calls as they relate to the assembly, and overall examine the code's interaction with the OS.

Looking for strings inside the executable is typically the first step most people will choose. For instance when reversing an executable one sees a URL for an unknown website or network address. This can be used to find a call-home or network communication portion of the code. Many times strings will bear a similarity to common formats such as email addresses. Once these are identified, any reference to them can be explored. Most commonly this will expose some of the email functionality of the program.

Finding the cross-references to commonly used system calls is another step in the process. By looking at things like any file access API, any files modified by the system can be traced. Likewise, other relevant APIs, such as network traffic and registry modifications indicate modification of the state of the system.

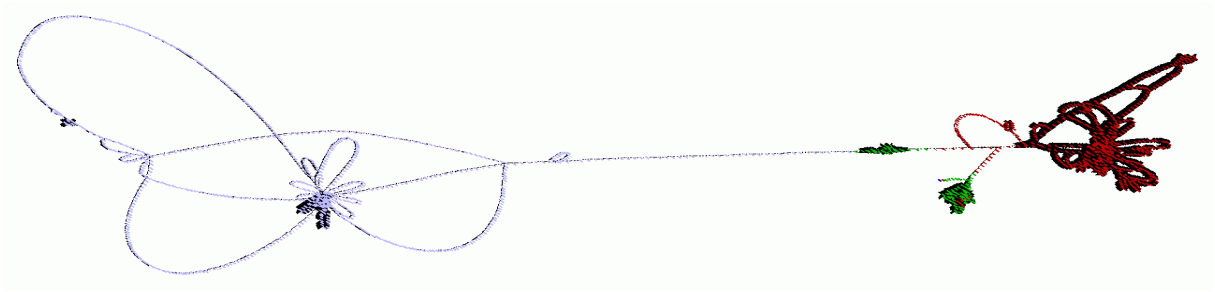

Figure 1: Visualization of the Netbull Virus Protected with the Mew Packer

This portion of the reverse engineering process is fundamentally imprecise and exactly what we want to improve upon. The common problems that beginners report is not knowing where to start within the code. Too often they are bogged down with the minute details rather than the big picture. VERA seeks to improve this by providing a high-level overview of the entire process execution.

4 VERA Architecture

Our tool consists of three main parts. First we have modified the hypervisor-based monitoring framework, Ether, to monitor and track program execution including memory reads and writes. Using the output of this data we construct a directed graph of all the basic blocks of an executable (represented as graph nodes). We use a weighted graph system from the Open Graph Display Framework (OGDF) to layout the graphs. The weight associated with each node is the number of times that block of code was executed in the analysis run. Similarly, edge weight is the number of times the particular control path was executed. The data is then displayed through the visualization interface. VERA provides a navigable interface to explore the code. It also links and connects to the IDA Pro reverse engineering tool to aid more detailed analysis.

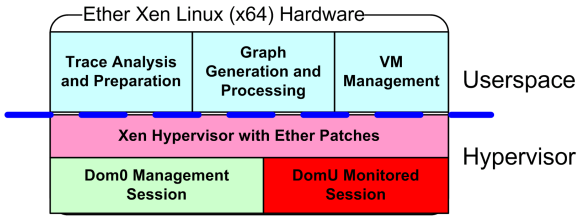

Figure 2: VERA and Ether Framework Architecture Modifications

4.1 Hypervisor Program Monitoring Using Ether

Ether is a set of patches and applications that have been added to the Xen hardware virtualization framework. The modifications alter Xen to be able to attach to and monitor a running program without detection. This gives us several distinct advantages when analyzing programs. First, program verification and protection code will not be triggered, thereby allowing the program to execute normally. Second, obfuscations that are meant to defeat traditional debugger and tracing systems are ineffective against the Ether system. Third, the overall state and structure of the virtualized system is preserved. Attempts by the program to detect monitoring will not yield results.

We have extended Ether in a couple of key ways to enable the analysis and visualization of applications, see Figure 2. Using the Ether control program, we added functionality that allows us to log reads, writes, and executes inside the program. This gives us the requisite data to generate our program flow graphs. Additional functionality determines the proper time to dump the current state of the running executable to allow us to circumvent any packer or obfuscation inside of the program. The initial implementation of Ether contained a hypervisor-based unpacking system. We have moved the logic of this to the dom0 user-space to enable use of more traditional software development tools. Finally, we have implemented a new import reconstruction tool to allow better analysis of DLL interactions.

The end result of these modifications is a trace file containing every statement execution, memory read and write, and the disassembly of the executed instruction. Periodic memory snapshots of the executable are also stored for further analysis using IDA Pro. The data is then processed for visualization and analysis.

4.2 Graph Parsing and Layout

VERA performs the following actions to parse trace files generated from Ether. First, the instructions are parsed to determine the location of the basic blocks. These basic blocks will create the nodes in our program's visualization. Transitions of execution between the basic blocks become the edges. A count of the number of executions determines the edge weight and is represented by thicker edge lines. Finally certain characteristics of the original executable are represented by altering the color of the associated node.

Once the graph is produced, we then use the Open Graph Drawing Framework (OGDF) to organize and layout the graph. Using this library allows us to render large graphs quickly and efficiently. Other systems such as the ubiquitous GraphViz could not handle graphs with large complexity. We chose the weighted symmetric layout option to organize the data. Other graphing methods such as circular layouts were found to not convey the appropriate information in an effective manner.

Once the layout is complete it is then sent to the visualization tool VERA.

5 Visualization and Presentation: VERA

The display engine uses a 2D view of the data which is translated into a 3D space. This provides for better zooming and introspection features for the code. It avoids many of the 3D aspects of representing the data. While 3D views generally provide a compelling view of the data, we have found that the 2D view is more useful for quick initial analysis. The graph data is represented in the OGDF GML format for ease of integration with that tool. This file format contains graph drawing primitives such as X and Y coordinates, along with coloring primitives. Our visualization tool parses and displays this data.

Each vertex of the code represents a basic block of execution of the program. This consists of all assembly operations that are contained between two adjacent branching operations. Many other programs choose to represent data at the function or method level of detail. The reason behind our decision lies with malicious software analysis. During the initial phase of execution, the program

does not follow the standard format of functions. Many of the obfuscations deliberately try to exploit this reliance of functions for analysis tools.

The characteristics chosen are the following: Yellow represents execution of code that is present in both the on-disk and in-memory executables. This indicates that the code is the same between the two executables. Red indicates execution in a section with high entropy. Most packers and obfuscators are able to compress an executable such that it has an even distribution of data. Areas of high entropy inside of the original executable indicate where the program has transitioned to the unpacked portions of the executable. Green is execution into non-existent code sections. If the executed instruction is non-existent in the on-disk executable, this indicates that the code is generated dynamically or is self-modifying. These data areas most often are dynamically allocated in heap space, such as that returned by malloc. Light Purple shows executiong where a section exists on disk, but not in the run-time executable This is most often found when data is allocated in the PE section headers, but not used until runtime. Neon green shows instructions that differ from the in-memory and on-disk executables. This is another sign that points to execution of self-modifying code.

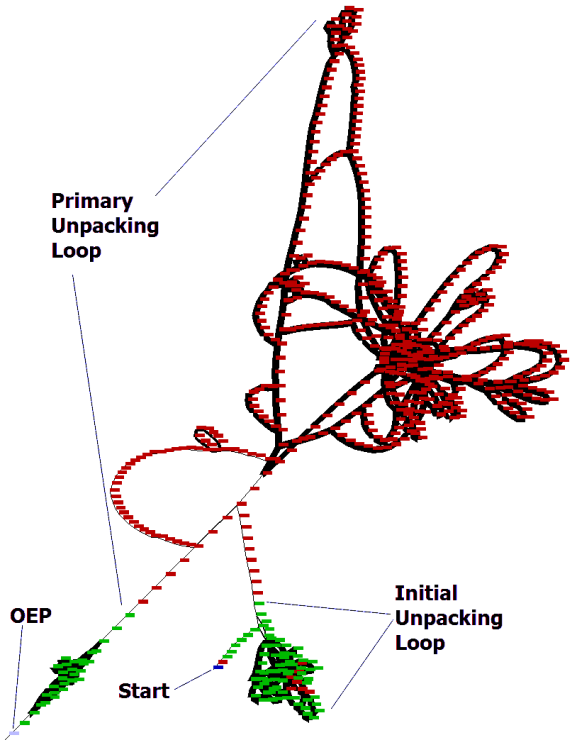

Figure 3: Close-up of the Mew unpacking loop

Once the data is presented the view can be manipulated to hone in on information. Zooming, panning, and interaction are all implemented in a method similar to the Google Maps interface. Zooming is accomplished by using the scroll wheel of the mouse. To navigate through the map, the left button of the mouse is clicked on the screen. This allows the entire display to be "dragged" to reveal different portions of the executable. Interaction with the data is done via mouse-over. When hovering over a specific node or basic block, information about it is displayed. This includes a link to bring up the data inside of IDA Pro, a partial disassembly, reference counts, and areas of memory modification. Right-clicking of the node allows labeling, which can then be propagated to IDA Pro.

The colors chosen could present trouble to someone with red-green color-blindness. To mitigate this an alternate set of colors are provided. These colors can be enabled with minimal effort by recompiling the program.

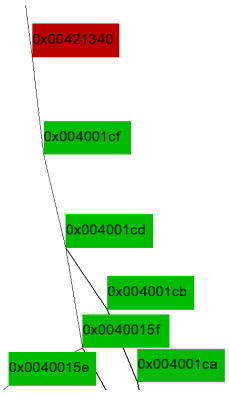

Figure 4: Zoomed detail view of the Mew unpacking code just after initial unpacking loop

By explicitly propagating information between IDA Pro and VERA, we are able to better facilitate analysis as the user can make annotations using either tool as discoveries are made and have those same annotations available in both tools. This supports better use of both tools and leverages both tools for faster reversing.

5.1 Feature Identification

Identifying program phases is broken down into a few discrete tasks: identifying the unpacking code, identifying the initialization portions, and identifying primary loops of execution. We break the analysis of programs down into these groups from empirical observation of many virus samples. Many non-malicious programs also exhibit this same behavior. These generalizations can be used in the majority of cases to identify the relevant behavior.

Identifying the unpacking loop of a program is relatively straight forward. It is typically any of the tightly bound loops found immediately after the entry point of the program. Figure 3 shows the unpacking loop for the MEW packer. The initial execution begins at the bottom with the starting point. From this figure, we see that there are multiple loops involved in deobfuscating the original program. Identifying the end of the unpacking loop is done by looking for the loop exits to continuous portions of solid colors. The longest contiguous portion of the executable is the light purple region. The first basic block of this region is most likely the original entry point of the program. This claim was validated by manually unpacking the sample and comparing the observed value with that of standard unpacking methods.

Initialization modules of the program are classified as long chains of basic blocks that have only one entrance and execution. In Figure 1 this is the middle portion that starts at the original entry point and ends at the left where the three branches occur. These portions of the executable typically deal with allocating memory, opening files and resources for later use, and accessing network resources. It is important to note that much of the initialization portion is not limited to this area and can be found in later portions. Finding the general area of initialization is a big step in narrowing down the scope of interest.

The final area of interest is composed of the main execution loops. In the center of the leftmost portion of Figure 1. The darker edges indicate heavily executed loops. By refining our analysis to this portion we can find the main backdoor portion of the system. This code activates itself, performs an initial call-back, and then waits for incoming connections.

6 Application of Analysis: Mebroot

To show the practicality of using VERA on a real data set, we have used VERA to analyze the initial loading point of the Mebroot trojan (MD5: 1f7fed180237ed352d274c69012a4717). Mebroot is a master boot record infecting malware that runs on a modern operating system [1]. Its primary purpose is to steal credit card numbers and other financial information from its victims. It also is used as a download agent to start other malicious code on an infected machine. Most of the malicious functionality is implemented in kernel mode. We will use VERA to analyze the usermode loading capability.

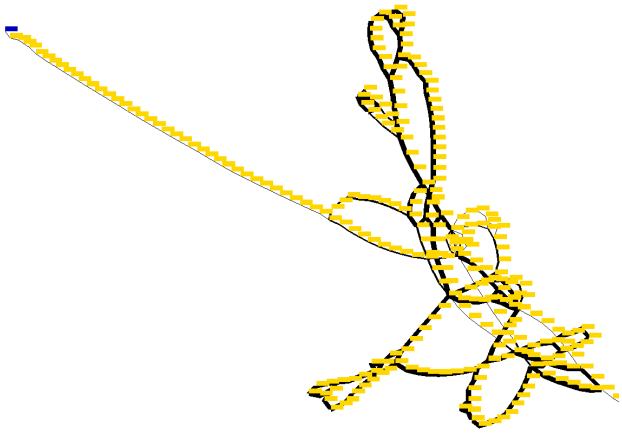

Figure 5: Mebroot initial 45 minute busy loop

The initial execution analysis was performed by letting Mebroot run for approximately 5 minutes. The results were shown in Figure 5 and were extremely limited. Since there is very little information shown, this portion of the program is most likely a busy loop to prevent rapid analysis and often leads the analyst down a faulty trail. By allowing execution to continue for 12 hours we were able to get a much broader view of the execution of the trojan. This is shown in Figure 6.

Figure 6: Mebroot overview of entire execution process

One of the difficult analysis tasks for Mebroot is its initial loading functions. The trojan prevents itself from being analyzed by first entering a busy loop for approximately 45 minutes. During this time, nothing of interest happens. Once this delay is complete Mebroot will then infect the master boot record of the host. The master boot record holds the initialization code which is later injected into the running Windows kernel after the boot process has occurred.

From the graph we were able to correlate the execution addresses to that inside of the disassembler IDA Pro. One of the main features we noticed is a technique known as mid-instruction point jumping. This obfuscation technique relies on the density of data in the Intel instruction set. When a static disassembler such as IDA Pro analyzes the code, the instructions are not the same as those that are executed. Nick Harbour discusses this very problem in [9] and is exactly what is present inside of Mebroot. Knowing this fact about Mebroot allows us to have IDA correct its view of the disassembled code. Knowing about this trick is extremely useful from the analyst perspective as it provides an accurate view of the code in question.

The rest of Mebroot executes inside of the priveleged kernel space and is not evident in this analysis. We know that the trojan successfully executed based on network logs and IDS signatures that identified the IP address as a Mebroot infection.

7 User Study

To evaluate how effective this tool and approach are for the analysis of executables, we have performed a preliminary user study. The users attended a reverse engineering training course that was given over the prior week. They learned the process outlined in section 3 and all passed a certification test based on this process. After users were given an introduction to the use of the tool and a set of printed instructions for VERA, they were walked through a typical analysis to familiarize them with the tool and approach. Following this training session, users were asked to perform an evaluation of the preliminary visualization tools of VERA for two malware samples. These samples were encrypted with two different packers: UPX and Mew.

Users were specifically asked to identify what aspects of their tools helped with each step in the standard evaluation process. First find the original entry point of a packed executable. Second execute the program to look for any discerning output. Use tools to monitor changes to the system while executing. Next load the program and begin the reverse engineering process, deobfuscating the binary as needed. Fourth identify the initialization portions of the program executable. Finally identify main loops to show relevant and interesting code sections to focus analysis.

At each stage in the process, users were requested to respond to how they had accomplished the main task for each step and to describe what had been discovered in that step.

At the end of the analysis phase, users were asked to evaluate the advantages and disadvantages of using VERA, whether using VERA sped up their analysis, and whether they found anything that that they did not think they would find using traditional techniques or missed anything they would have found using traditional techniques. Finally, users were asked whether they were likely to use VERA again and whether they would recommend VERA to colleagues.

The user success in finding specific sections of code are shown in Figure 7, where the number of users who found the original entry point (OEP), initialization code, and main loops are displayed. Further, as shown, all users said that they were likely to use VERA again and would recommend it. This indicates acceptance for VERA from both novice and experienced users, although a substantially larger experiment would be necessary for statistical significance.

The only substantially negative comment of all user responses

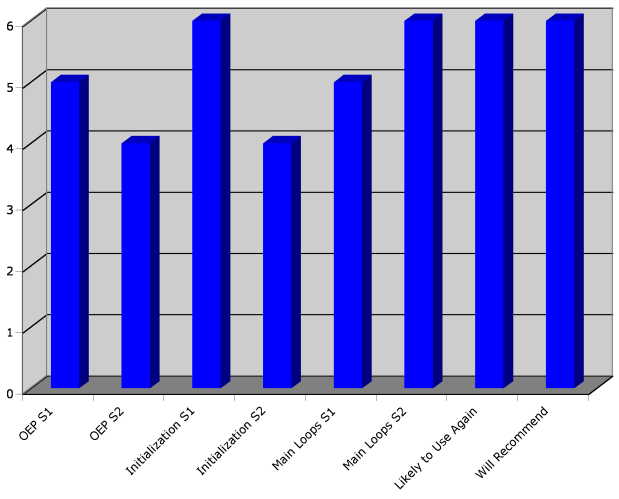

Figure 7: This chart shows the number of users who found specific sections of code for the two different samples using VERA. OEP stands for the original entry point. The chart also shows that all six users said they would use VERA again and that they would further recommend VERA.

.

was that although User 1 was able to identify the beginning of a loop, that user could not identify the end of the loop; this user said the many loops became convoluted.

User 1 said he was "able to pick out areas of interest more quickly". User 2 "saw tight main loops at a glance, much easier than using IDA alone". User 5 said it was "easy to identify original entry point through visual representation of execution paths". User 6 said "its great to be able to see where the important stuff that actually executes a lot".

Some significant suggestions were provided for improving VERA. User 2 would "like to be able to enter a memory address and see basic blocks that reference the address highlighted". User 3 would like the ability to "hide and show all or individual loops", which was similar to a suggestion by User 6.

In the overall evaluation, User 1 said "Wonderful way to visualize analysis and to better focus on areas of interest", User 2 said "Fantastic Tool. This has the potential to significantly reduce analysis time", and User 4 said "It rocks. Release ASAP". Overall, this user study indicates both the usefulness and usability of VERA.

8 Conclusion and Future Work

The VERA framework we have presented provides an enhanced method to speed reverse engineering. This tool has been used in a variety of commercial and academic settings to better understand the flow and composition of a compiled executable. The user study shows that the tool enhances analysis and lowers the total amount of time necessary to reverse engineer an executable.

We have modified the Ether analysis framework to better enable traditional analysis techniques, including our own visualization tool. These have been added to the mainline Ether system.

There are several areas of future work that will be explored. First, better highlighting of the loops inside the executable will be implemented. This was a common request from users for the tool. Second, a kernel based method of program analysis will be developed. This is in response to the transition of modern malware to the Windows kernel architecture. Finally a 3D visualization environment will be explored to provide further insight into program analysis.

Acknowledgements

The authors wish to thank Alan Erickson, Cort Dougan, Paul Royal, Artem Dinaburg, and Moses Schwartz for their invaluable help.

References

[1] K. Alkio. Mbr rootkit, a new breed of malware. F-Secure Blog. http://www.f-secure.com/weblog/archives/00001393.html.

[2] U. Bayer. Ttanalyze: A tool for analyzing malware. Masters thesis, Technical University of Vienna, 2005.

[3] J. Bohnet and J. Döllner. Visual exploration of function call graphs for feature location in complex software systems. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 95–104, New York, NY, USA, 2006. ACM.

[4] C. Cifuentes, T. Waddington, and M. V. Emmerik. Computer security analysis through decompilation and high-level debugging. In *Eighth Working Conference on Reverse Engineering*. IEEE Computer Society Washington, DC, USA, 2001.

[5] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2008.

[6] P. Ferrie. Attacks on virtual machine emulators. *Symantec Advanced Threat Research*, 2006.

[7] P. Ferrie. Anti-unpacker tricks - part one. *Virus Bulletin*, 2008.

[8] F. Guo, P. Ferrie, and T.-c. Chiueh. A study of the packer and its solutions. In *RAID*, Cambridge, Massachusettes.

[9] N. Harbour. Advanced software armoring and polymorphic kung-fu. In *Defcon 16*, Aug. 2008.

[10] HBGary. Responder professional. Product Description Page. https://www.hbgary.com/products-services/responder-professional/.

[11] Hexrays. Ida pro disassembler and debugger. Product Description Page. http://www.hex-rays.com/idapro/.

[12] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM)*, Oct. 2007.

[13] J. Ma, J. Dunagan, H. J. Wang, S. Savage, and G. M. Voelker. Finding diversity in remote code injection exploits. In *Internet Measurement Conference*, Rio de Janeiro, Brazil, 2006. ACM.

[14] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, San Diego, CA, 2007. ACM.

[15] D. Quist and V. Smith. Covert debugging: Circumventing software armoring. In *Blackhat USA*, Aug. 2007.

[16] P. Royal. Alternative medicine: The malware analyst's blue pill. In *Blackhat USA*, Aug. 2008.

[17] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware, 2006.

[18] J. Stewart. Ollybone: Semi-automatic unpacking on ia-32. In *Defcon 14*, Las Vegas, NV, 2006.

[19] A. Telea and L. Voinea. An interactive reverse engineering environment for large-scale c++ code. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*, pages 67–76, New York, NY, USA, 2008. ACM.

[20] A. Vasudevan and R. Yerraballi. Spike: Engineering malware analysis tools using unobtrusive binary-instrumentation. volume 48, Hobart, Tasmania, Australia, January 2006. Australian Computer Society, Inc.

[21] Y. Xia, K. Fairbanks, and H. Owen. Visual analysis of program flow data with data propagation. In *VizSec '08: Proceedings of the 5th international workshop on Visualization for Computer Security*, pages 26–35, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] O. Yuschuk. Ollydbg debugger and disassembler. Product Description Page. http://www.ollydbg.de/.

[23] Zynamics. Binnavi. Company Product Description Page. http://www.zynamics.com/binnavi.html.