



ARCHITECTURE OF MALWARE TRACKER VISUALIZATION FOR MALWARE ANALYSIS

¹CHAN LEE YEE, ²MAHAMOD ISMAIL, ³NASHARUDDIN ZAINAL, ⁴LEE LING CHUAN

^{1,4}PhD Student, Department of Electrical, Electronic and System Engineering, Universiti Kebangsaan Malaysia, Malaysia

²Professor, Department of Electrical, Electronic and System Engineering, Universiti Kebangsaan Malaysia, Malaysia

³Dr, Department of Electrical, Electronic and System Engineering, Universiti Kebangsaan Malaysia, Malaysia

E-mail: [1chanleeyee@f13-labs.net](mailto:chanleeyee@f13-labs.net), [2mahamod@eng.ukm.my](mailto:mahamod@eng.ukm.my), [3nash@eng.ukm.my](mailto:nash@eng.ukm.my), [4lclee_vx@f13labs.net](mailto:lclee_vx@f13labs.net)

ABSTRACT

Malware is a man-made malicious code designed for computer destructive purposes. The early destructive programs were developed either for pranks or experimental purposes. However, in this day and age, malware are created mainly for financial gain. Since years ago, the use of malware attack tools, such as keylogger, screen capture software, and trojan were rapidly used to commit cybercrimes. The figures are expected to increase significantly and the attack tools are becoming more sophisticated in order to evade the detection of current security tools. The malware debugger analysis process is an essential part of analyzing and comprehending the purpose and the destructive part of the malware. It is an exhausting and time consuming task; moreover, in-depth computer knowledge is required. With the popularity and variety of malware attacks over the Internet, the number of virus needed to be analyzed by computer security experts are rapidly increasing and has bottlenecked the effectiveness of the analysis process. In this paper, we present a method to visually explore the reverse engineering of a binary executable flow over time to aid in the identification and detection of malicious program on x86-32 platform. We first achieve the pre-execution analysis for a sketch of a program's behavior by combining static analysis and graphical visualization to construct a control flow graph (CFG) as an interface for the analyzed code. Each node in the CFG graph which represents a basic block allows analysts to be selective in the components they monitor. All nodes in the CFG express the complex relationships and causalities of the analyzed code. As the binary executes, those codes that are dynamically generated will be monitored and captured; thus, a fuller understanding of the execution's behavior will be provided. The backward track approach which allows analysts to restudy the changes of the executed instructions' memory during dynamic analysis provides a chance for analysts to restudy the execution behavior of the executed instructions. The overall architecture of the visualization debugger, both statically and dynamically will be explained in this paper. To the end of the paper, we analyze a malware test case; W32/NGVCK.dr.gen virus with our malware tracker visualization toolkit and the analysis results proves that our visualization malware tracker tool can simplify the analysis process by displaying the analyzed code in basic block approach. This work is a substantial step towards providing high-quality tool support for effective and efficient visualization malware analysis.

Keywords: *Malware Visualization Debugger, Static Analysis, Dynamic Analysis, Malware Analysis*

1. INTRODUCTION

In computing, a software application is engaged to instruct computers to perform the indicated task designed, either for benign or destruction purposes. With the intention of concealing the design methodology and protect the privacy of an application, many software are ultimately translated into binary before execution. Binary is much harder to understand compared to high level scripting

programs as only zero (0) and one (1) are represented inside the binary code. However, the advantage of a binary executable can be misused for malicious purposes where the destructive code can be distributed either in a dedicated binary executable file or hidden inside the victim's binaries. The creation of malware as a primary vehicle for carrying out various cybercrimes for huge financial gains has become today's most serious security threat on the Internet. According to



Computer Economics 2007 Malware Report, malware infections in 2006 cost \$13.3 billion dollars. Although the trend over the last two years has downturned the cost of malware infections, the cost of malware infections should still be a concern to companies of any size. The report states two factors for the reduction in malware infections cost; the wider spread deployment of anti-malware applications, and malware targeted at specific organizations and people [1].

Although many automated security tools have been created to automate the detection of malicious portions of a program, unfortunately, the tools are still not smart enough to track down new protection techniques that are created to dodge security software. With the proper skills and knowledge, the old school method of malware reverse engineering techniques remains the most effective way to distinguish the evil code from a benign program.

Malware tracking is a difficult and time consuming process which provides insight of the structure and functionality of an executable program. Currently, static and dynamic reverse engineering tools are available to help security researchers analyze processes to verify whether any irregular code is hiding inside a test program [2]. In static analysis, a further insight of the malware body is studied [3]. It parses the instructions that are found in the binary image to understand and detect the malicious functions and its shell code. String searching tools and disassemblers (e.g. IDA Pro [4]) are examples of static analysis tools. Dynamic analysis is designed to inspect and monitor a run-time executable action [5]. It identifies the execution instructions and the behavior via monitoring the execution. The changes of the system including registry modification, installation of new service and network communication will be kept track. The run-time of an executable is controlled with dynamic analysis tools. It includes debuggers (such as OllyDbg [6], GDB [7], WinDbg [8]), Operating System State tracking (such as Sysinternals' Processmon tool [9]) and system call. Most of the executable debugger tools generate a large swath of assembler instruction code which is arranged in ascending order from the smallest to the largest of offset address. The parsed assembly code could be a used or unused code. Thus, more time has to be invested in comprehending a binary executable program.

This paper aims to illustrate that the visualization of malicious or vulnerable program data flow tracking can facilitate a security expert in their investigation for purposes of comprehending the

irregular activities of a malicious program and creating signatures for security devices for automated detection. The targeted suspicious program will be analyzed and parsed into human readable assembly code sequences. The assembly program will interact with graphical visualization and display the analyzed code in basic block approach. The control flow graph information approach simplifies the identification of malicious program instruction in fraction code. The dynamic analysis module is devised to work with graphical visualization to inspect and visualize the execution path. The approach can provide the overall concept of program execution in a particular variable, register value and memory location.

In summary, this paper is to demonstrate the ability to develop a competitive visualization of malicious binary code tracking and analyzing in a much simpler way. The overall architecture of the malware tracker visualization toolkit will be discussed in this paper. We address the combination of static and dynamic techniques with a visualization flow chart creator to construct and maintain the data flow analysis. Key features of our approach are the ability to update the analysis to include overwritten code and the ability to store the changes of IA-32 bit 4GB memory for the backward tracing purpose. Towards the end, we make several contributions. We propose and develop the malware tracker visualization with the integration of static and dynamic debugging process to simplify the malware reverse engineering process. With the combination of static and dynamic analysis, security analysts are able to find and analyze code that is beyond the reach of either static or dynamic analysis alone, thereby providing an in-depth understanding of a suspicious creature's possible behavior. The proposed visualization graph program flow gives an analyst a more comprehensive view of an executable behavior. Irregular events are more intuitively identifiable when presented visually. The integration of a dynamic debugging tool with the visualization graph enables the sophisticated assembly code distribution steering and animation as well as visualization. The proposed backward trace done by storing the changes of IA-32 bit memory and instruction's contents into a database allows analysts to review or restudy the executed instructions over a large swath of binary code without having to restart the dynamic analysis process.

This article is articulated according to the following structure. Section 2 describes related

work. An overview of architecture engine is presented in section 3. Section 5 discusses the experimental results of the malware tracker visualization toolkit. Finally, section 5 briefly concludes and outlines future work.

2. PREVIOUS AND RELATED WORK

Application of the debugger is to run and monitor the execution of an arbitrary program. It can be used to alter or control the execution of a target program in order to monitor the memory and variable of the registers. Developing a perfect debugger that translates Windows/x86 binaries into assembly instructions is a difficult and complicated task as many considerations need to be taken into account. For example, variable size of instructions, the presence of data inside the code section and the hidden portion of code that is unreachable statically. As illustrated in [10], a hybrid approach that integrates control flow with linear traversal techniques is used to improve the coverage of translation and reduce the disassembler errors. To further increase the accuracy of analysis, disassemblers implement speculative disassembly techniques [11] that verify the disassembler results after a certain assumption is made to continue the analysis process in order to accept the translating results. For instance, Kruegel et al. [12] use control flow graph analysis and statistical methods to increase the accuracy of producing valid disassembled instructions.

Visualization of program execution to study and monitor program executions have been used in the past with good results. Xia [13] presents their methodology to visually represent and analyze the program flow of a system. With the proposed visualization, users are able to detect irregularities in binary execution and accentuate trouble spots of illegal file access. The taint propagation gives the user the ability to gauge the impact of a potentially malicious program or file to aid in the recovery process.

Madou et al. [14] combined static and dynamic techniques to identify unreachable code that is possible through either technique alone. They start from an execution trace and construct a control flow graph of a program to thwart software resistance techniques; thus, additional code will be able to be found and located.

A Windows's instrument, BIRD: Binary Interpretation using Runtime Disassembly [15], translates the binary file into individual assembly language instructions via disassembly both

statically and dynamically. It works well on compiler generated programs. By integrating static and dynamic disassembling, BIRD is able to locate the unknown area as much as possible. Unfortunately, the instrument does not come with any debugger information such as the symbol table, relocation table, etc.

The VERA framework [16] presents a dynamic analysis method to visualize the overall flow of a program. It provides an enhanced method to speed up the reverse engineering process in order to provide better understanding of the flow and composition of a compiled executable. The authors claim that the tool is able to reduce the amount of time to extract key features of an executable and improve productivity.

3. MALWARE TRACKER VISUALIZATION ARCHITECTURE OVERVIEW

3.1 Overview

Malware reverse engineering is a process that can be tedious and very time consuming. It requires a lot of patience in understanding the function and the true intent of a program. The proposed approach of the visualization malware tracker by combining static and dynamic techniques to construct and maintain data flow analysis that form the interface simplifies the overall malware analysis process; thus, the true intention of the creature is understandable in a comprehensive way.

Figure 1 illustrates the architecture of the malware tracker visualization toolkit. Typically, the toolkit consists of a Mini Graph that visualizes the analysis results from both static and dynamic analysis. The visualization results generated from Mini Graph provide a comprehensive view in regards to the path of execution program to the analysts. Unlike the traditional analysis and debugger applications, the toolkit is integrated with a database to store the entire data changes of every execution in memory. This is to allow the analysis and debugger processes to be able to resimulate and restudy the changes of memory status of the executed instructions via the Backward Tracing function. Overall, the malware reverse engineering tasks via the Malware Tracker Visualization consists of the following steps:

- 1 Execute the suspicious program in an isolated environment
- 2 Attach the suspicious program to be analyzed via the Visualization Debugger Interface

- 3 Extract the targeted Win32 portable executable (PE) program via the Debugger Engine to handle the entire reverse engineering process.
- 4 Disassemble the targeted program via the Disassembler Engine component to translate the machine language into human-readable assembly language.
- 5 Analyze the targeted program statically and present the analyzed results in the Disassembly Windows (instructions in assembly code listing) and Mini Graph (control flow graph (CFG) based).
- 6 Analyze the targeted program dynamically and work simultaneously with Mini Graph. Every stepping execution happening on the Disassembly Windows will interact with the Mini Graph for the overall execution paths.
- 7 Resimulate the traced execution based on the collected trace information deterministically via the Backward Tracing component.

isolation and snapshot feature, where it enables the analyst to restore the operating system back to its original pre-infection state in a separated environment. This is a precaution step to ensure no normal computer activity is compromised.

The configuration of the virtualization system to have the ability to take the current state of the system is crucial and has been set as our necessary baseline for our malware analysis environment. The current state provides a known-good system to compare with subsequent system state over the execution of suspicious binary. Once the snapshot feature is taken, the subsequent system state can recover to its pre-infected state in a very short period of time without re-installing or re-configuring the environment.

3.3 Visualization Debugger Interface

As illustrated in Figure 1, the toolkit is built as a layered system. The user interface is implemented using the Python programming language and the Qt application development framework [19] for flexibility. Figure 2 shows the graphical view of the proposed high level debugger. The left side of the figure shows the Disassembly Windows of the Visualization Debugger Interface divided into four columns, “Address”, “Hex”, “Disassembly” and “Comment”. The column of “Address” illustrates the instruction’s address in the memory. Both operation code (opcode) and assembly language are located at column “Hex” and column “Disassembly”. The “Comment” column is an optional column and the data that appears in the column is limited to the Application Programming Interface (API) detected. The right side of Figure 2 displays the equivalent of general purpose registers values. The general purpose registers is much like a variable in any other high-level programming language. It acts as temporary holders for values. The Info section, located at the bottom contains the current segment register and other related information.

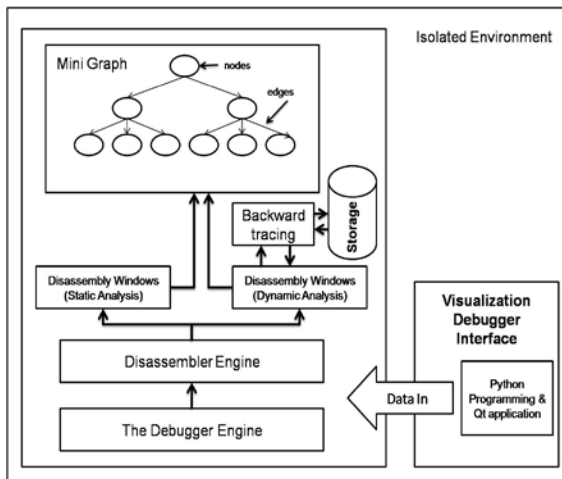


Figure 1. System Architecture of the Visualization Malware Tracker Toolkit

3.2 Isolated Environment

Reverse engineering of an unknown binary executable is a tiresome and repetitive process. With no prior knowledge of testing a piece of software, various situations might be encountered and the worst situation is one in which the operating system’s functionality or safety when analyzing a suspicious instance could be jeopardized. Thus, a well configured and isolated environment [17] is crucial to protect the system from any nefarious activity that could happen while analyzing or studying an unknown creature. In this project, VMware software [18] is chosen as our isolated virtualization system. The selection is due to the

will be logged together with the time of each activity. This approach allows analysts to trace back their analysis activities for recording and reporting purposes.



Figure 2. The Visualization Debugger Interface

3.4 The Debugger Engine and Disassembler Engine

The Debugger Engine is designed for examining and manipulating debugging targets on the x86/Windows platform. Analyzer can set breakpoints, monitor events, read memory processes and view the analyzed program graphically. As illustrated in Figure 1, the Visualization Debugger Interface will emit the targeted creature to the next phase of the Debugger. The “Data In” refers to the targeted executable file that needs to be analyzed.

The Disassembler Engine is designed to translate the machine language code to high level readable assembly program. Only the Win32 portable executable (PE) files that targets the Intel x86 instruction set is available to be translated. Before initiating the static and dynamic analysis components, the Disassembler Engine will process the entire output from the Debugger Engine to ensure that the output can be sent to either the Static or Dynamic Analysis component. The details of the Static and Dynamic Analysis will be discussed in the next section.

3.5 Static and Dynamic Analysis

Both static and dynamic analysis modules are designed to interact with the debugger engine to conduct static and dynamic reverse engineering, respectively. The static analysis enables security analysts to find and analyze binary codes by traversing the statically analyzable control flow which begins from known entry points in the code. The initial statically analyzable code may be incomplete because some codes are not reachable

through statically analyzable control flow, and may lead to un-analyzed code.

Dynamic analysis analyzes and locates analyzed code that is missed by static analysis. It keeps track of already translated instruction block and checks whether the code has been modified every time it is executed. Instrumentation [20] is a technique whereby an existing code fragment is modified by adding small code snippets at key points in order to change its behavior. Accurately detecting new un-analyzed code is important, as it allows analysts to re-instrument the new codes and use them to re-seed the new parsing results.

The integration of static and dynamic approach is to ensure the analysis results have zero room for disassembly errors. In the effort to ensure the accuracy of the analysis output, the framework applies both static and dynamic disassembly. The analysis framework begins the analysis process statically to reveal as many instructions as possible, and these instructions will be marked as *translated areas, P*. The rest of the instructions that can only be uncovered during dynamic analysis will be marked as *unknown areas, P'*. The *unknown areas* will be revealed gradually at run-time during dynamic analysis. This approach allows the translating results to be returned as much as it can especially when the program's control is transferred to the *unknown areas*. By integrating static and dynamic analysis, the parsing process of every instruction in the targeted binary is guaranteed.

3.6 Forward and Backward Tracing

As the binary executes, new dynamic code will be generated. For every dynamic analysis, we determine the extent code which has been overwritten. If code is overwritten, our visualization debugger will clean up the existing CFG and re-invoke the parsing process to update the CFG of the program. The re-invoke process includes identifying the new code and presenting the updated CFG to analysts.

Throughout the dynamic analysis, the memory changes of dynamic analysis will be monitored. Any changes of the execution memory will be saved in our database. As mentioned earlier, reverse engineering is a time consuming process and can be tedious. Some important parameter or relevant register's value might easily be overlooked. Sometimes, the understanding of previous execution, could lead to better comprehension of the next execution; thus, it is crucial for any debugger to provide forward and backward track

functions in dynamic analysis. Forward track function allows stepping execution and monitors the changes of both register and variant. Backward track function allows analysts to track back executed instructions. Previous executed instructions can be restudied and re-understood for the next forward execution functions.

3.7 Mini Graph - Visualization Results

Throughout the reverse engineering process, both static and dynamic analysis will communicate with uDraw(Graph) software [21] which is installed in the Mini Graph to display the analysis results in control flow graph (CFG) based. uDraw(Graph) is a freely available package from the University of Bremen, Germany for creating flow charts, diagrams, hierarchies or structure visualizations using automatic layout. We use the software to create a graph representation of the analysis module which visualizes the result of the analyzed code graphically and simultaneously with static and dynamic analysis process. The visualization debugger which acts as a socket client, is used to send the transmitted graph commands to the Mini Graph as socket server to instruct the uDraw(Graph) to visualize the assembly command in basic block base.

The overall idea of the approach is that the static analysis engine will emit disassembler control flow to Mini Graph to display the interactions among collaborating objects in sequence of basic block diagrams. A key feature is to determine the skeleton of an analyzed executable file. Static analysis and control flow graph forms an analysis interface that simplifies the analysis task, providing a flexible analysis mechanism. The integration of static analysis and control flow graph allows analysts to be selective in the components they analyze. The analysis operation could be performed based on the components that they have selected, in the granularity of data collected.

The Mini Graph presents several different algorithms for positioning nodes and routing edges, and this is extremely useful information for analysts when dealing with larger functions with many conditional jumps. It begins by dividing the assembly code using the basic block approach which is designed to analyze code independently. The basic block approach forms a contiguous block of code with a single entry point and a single exit point at both the beginning and the end of the block without any jumps or jump targets in the middle. The control transfer instruction (CTIs) [12] such as conditional and unconditional jumps or return

instructions will control the connection of every basic block to construct a visualization graph. Instructions such as je, jne, and jmp are the example of instructions that are grouped under jump conditions.

The analysis process can be done by traversing the statically analyzable control flow starting from known entry points of the code. It is very common to analyze a program with varied circumstances. As soon as an instance executable file is parsed with static disassembler process, an overall structure of the body program including the designed task with different condition path will be performed in a list of translated assembly program format. Unfortunately, the static analysis process fails to trace the program's execution path dynamically. Thus, the direction of instructions' path fails to be determined.

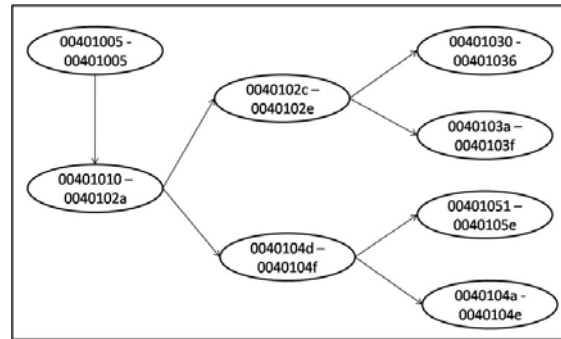


Figure 3. A Mini Graph debugger

The reverse engineering of sequence visualization through static analysis is the next logical step for the tool. A visualization graph generated using an example function is shown in Figure 3. The nodes in this figure represent basic blocks and are labeled with the start address of the first instruction and the end address of the last instruction in the corresponding instruction sequence. The solid and directed edges between nodes represent the target of control instructions. In this example, the algorithm is invoked for the function start at address 00401005 and a jump candidate, 00401010. Conditional branches will be handled after the second node, which are 0040102c or 0040104d. In particular, the option of the execution jump candidate is relied on the return result of conditional jumps at node 00401010 during dynamic analysis. However, static analysis only gives an overall structure or idea of the analyzed program. Figure 4 shows the results of the analyzed program generated from the Static Analysis displayed at the Mini Graph component.

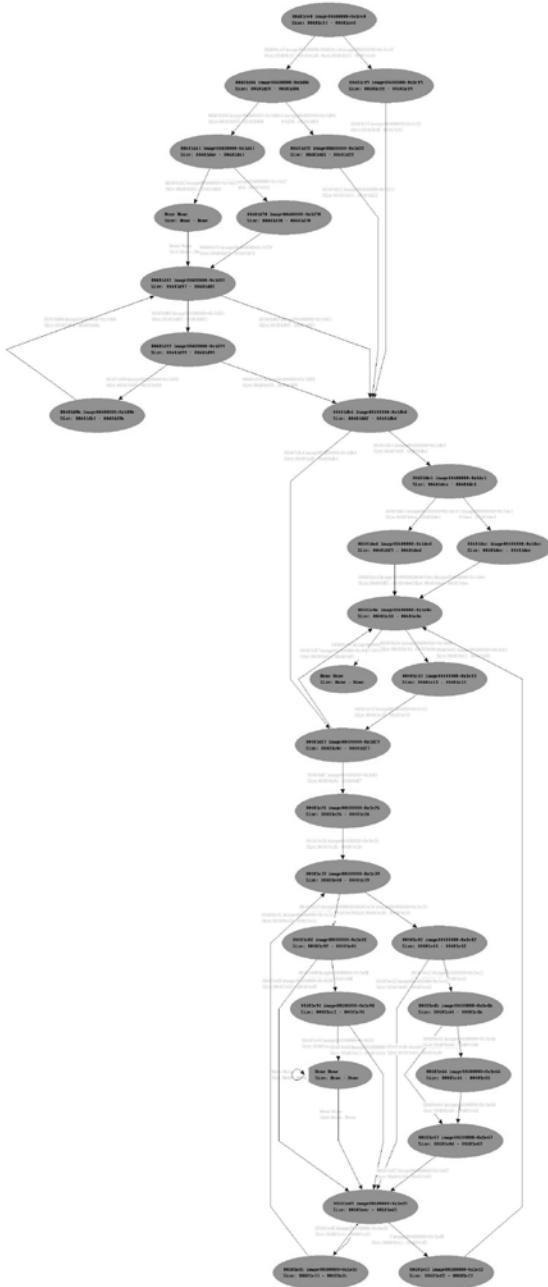


Figure 4. Visualization Results

The dynamic analysis works together with the CFG graph generated by Mini Graph. Every debug option generated at the Visualization Debugger Interface will communicate with Mini Graph. An execution instruction at a particular node will be highlighted to notify the analyst of the current position of the overall debugging process and also to show the analysts the overall execution path of the analyzed program based on certain environment.

The static analysis examines a program code statically without executing a program. Unfortunately, due to the statically analyzable code, some analyzed results will not be generated until run-time and the un-analyzable code is only reachable via dynamic analysis technique. Dynamic analysis is designed to monitor and visualize the execution path of an executable. It enables the tracing and stepping through an instance program. The entire intermediate values of variables will be monitored as well. To initialize the dynamic analysis, single stepping is executed and the execution will return to the control debugger to wait for further instructions. The entire dynamic process will be preceded via single stepping and this process will keep routine until the end of the process or the analysis task is terminated.

The idea of this section is to trace the executable program by running step by step processes together with the Mini Graph generated at the previous static analysis section. The relevant portion of the code will be highlighted to display the unification procedure while the program is stepping through. The idea of this approach is to provide an overall concept of the program execution in a manner that shows the paths of control flow. The intermediate values of the parameters involved in the program can help the security analyzer to understand accurately the details of the suspicious program execution.

Figure 5 shows the interaction of the dynamic debugger with Mini Graph. Every stepping execution on the left will interact with mini-graph simultaneously as shown on the right of the figure and the relevant ellipse shape will be highlighted via the execution stepping program passing through the portion of instructions. For detailed instructions set within each basic block, analysts can simply click the desired ellipse and the detail of instructions will be displayed.

Another important feature of dynamic analysis is the capability of updating the disassembler and control flow graph. As mentioned previously, some analyzable code is beyond the reach of static analysis until the execution in real-time. Thus, in order to gain better understanding of the malicious code, dynamic analysis is designed to analyze the code overwrite [22] at real-time. Dealing with code overwrite is complicated as some new code is not presented until the code pointer is pointed to the address. The intention of code overwrite is to invalidate portions of an existing static code analysis.

As the binary executes, new dynamic codes will be generated. For every dynamic analysis, we determine the extent to which the code has been overwritten. If the code is overwritten, our visualization debugger will clean up the existing CFG and re-invoke the parsing process to update the CFG of the program. The re-invoke process includes identifying the new code and presenting the updated CFG to analysts.

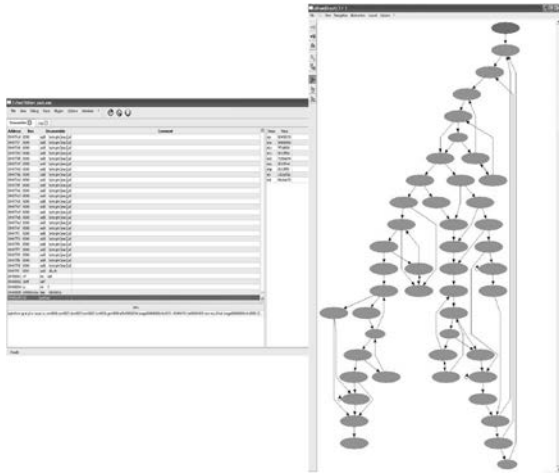


Figure 5: Interaction of dynamic debugger with mini-graph

Throughout the dynamic analysis, the memory changes of dynamic analysis will be monitored. Any changes of the execution memory will be saved in our database. Since reverse engineering is a time consuming process and can be tedious, some important parameters or relevant register's values might easily be overlooked. Sometimes, the understanding of previous execution could lead to better comprehension of the next execution; thus, it is crucial for any debugger to provide forward and backward track functions in dynamic analysis. Forward track function allows stepping execution and monitors the changes of both register and variant. Backward track function allows analysts to track back executed instructions. Previous executed instructions can be restudied and re-understood for the next forward execution functions.

4. EXPERIMENTAL RESULTS

In this section, we present some experimental results. Since our toolkit is in an early stage of development, our analysis can only analyze the Windows binary in user mode.

4.1 Original Entry Point Identification

As mentioned previously, the integration of the Debugging Engine with the Mini Graph provides a solution of identifying the original entry point (OEP) of a packed executable. This approach is helpful in removing packers which were implemented by many malware samples. Typically, packer or obfuscation technique is a common manner that has been implemented by malware authors to thwart the detection of computer security tools. To perform an evaluation of the preliminary via our malware tracker visualization framework, a binary executable file with the original entry point, "00401005", encrypted with the BeRoEXE Packer (BEP) was loaded with our framework. Figure 6 shows the translated results transmitted to Mini Graph. Referring to the figure, a deobfuscating loop was involved. The generated CFG results in Mini Graph being able to locate the OEP easily by selecting the node with only incoming generated node. As shown in the figure, nodes within the program were correlated with each other. Typically, the nodes that are most likely to consist of the OEP of a program is the node with only incoming generated node and only two nodes comply to the condition, which are nodes labeled as 1 and 2. Both instructions within the node labeled as 1 and 2 are shown in Figure 7 (a) and Figure 7 (b). The instructions with the node labeled as 1 in Figure 6 illustrate the instruction of jumping out of the BEP obfuscation loop to the offset address of the original entry point at 00401005.

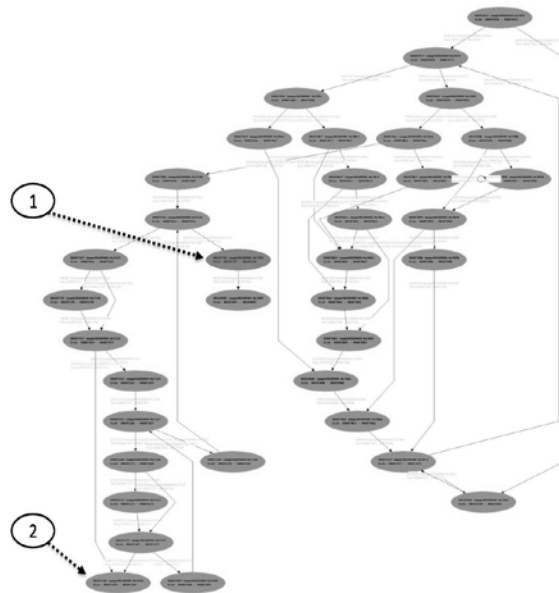


Figure 6: Close-up of the BeRoEXEunpacking loop


```
(a)
00407196  popad
00407197  jmp  image00400000+0x1005 (00401005)

(b)
00407194  popad
00407195  ret
```

Figure 7.(a) Execution instructions that instruct the binary to jump out of the obfuscation loop to the original entry point of the binary at the offset address of 00401005

Figure 7. (b) Execution program finish

4.2 Forward and Backward Experimental Test

In this section, the forward and backward tracing component will be exhibited. Both the top and the bottom of Figure 8 represent the forward and backward tracing, respectively. For every instruction executed, the register values of the particular instruction will be shown at the right hand side. In this experiment, the top of the figure shows an executable file that implements the BEP packer which has been successfully unpacked. It was located at the real executable body of the program and landed at the 00401017 address. The particular instruction will be highlighted at the Disassembler section. An analyst can use the backward tracing component to travel back to previous instructions to restudy the memory stage of previous instructions. The bottom of the figure shows the instruction's memory stage that has travelled back to instructions within the BEP packer loop. As shown at the bottom of Figure 8, nothing was changed at the Disassembler section and the offset address, 00401017 maintained highlighted. However, the eip address at the right hand side has changed to offset address at 00407060, and the particular instruction in assembly language was updated as shown in the Info section.

4.3 Malicious Code

In this section, the malware tracker visualization is used to analyze malware for its effectiveness test. A real world malware, named as W32/NGVCK.dr.gen by McAfee antivirus software, is chosen as our test creature and its malicious behavior will be analyzed.

Figure 9 shows the destructive code fragment of the W32/NGVCK.dr.gen virus, together with explanations of some instruction codes. The fragment program explains the attack method that was implemented by the malware. Throughout the code, we can conclude that the malware

implemented the appending virus infection technique [23] targeted at Windows PE executable files. As illustrated by the instruction code in Figure 9, the virus added a new section header at the end of the section PE table and place the virus body in that section by modifying the NumberOfSection field of the PE header. The virus program was added at the end of a file, and then turns control over to execute the virus program before the instructions of the original program was executed.

```
00401235  mov  dword ptr [esi+4C], 636816E
0040123C  mov  dword ptr [ebp+40140E], esi
00401242  xor  eax, eax
00401244  ax, word ptr [esi+6]
00401248  mov  dword ptr [ebp+401493], eax
0040124E  inc  word ptr [esi+6]
00401252  mov  eax, dword ptr [esi+28]
00401255  add  eax, dword ptr [esi+34]
00401258  mov  dword ptr [ebp+40140E], eax
0040125E  mov  eax, dword ptr [esi+50]
00401261  mov  dword ptr [esi+28], eax
00401264  mov  dword ptr [ebp+401403], eax
0040126A  add  eax, 1000
0040126E  mov  dword ptr [esi+50], eax
00401272  add  esi, 0E8
00401278  mov  eax, 28
0040127D  mov  ecx, dword ptr [ebp+401493]
00401283  mul  ecx
00401285  add  esi, eax
00401287  mov  dword ptr [ebp+40140E], eax
0040128D  mov  dword ptr [esi], 6C796368
00401293  mov  eax, 1000
00401298  mov  dword ptr [esi+8], eax
0040129E  mov  eax, dword ptr [ebp+401403]
004012A1  mov  dword ptr [esi+C], eax
004012A4  mov  eax, 9F5
004012A9  mov  dword ptr [esi+10], eax
004012AC  mov  eax, dword ptr [ebp+40140E]
004012B2  mov  dword ptr [esi+14], eax
004012B5  mov  eax, E0000020
004012BA  mov  dword ptr [esi+24], eax
004012BD  mov  edi, dword ptr [ebp+40147E]
004012C3  mov  eax, dword ptr [ebp+40140E]
004012C9  add  edi, eax
004012CB  mov  esi, <ModuleEntryPoint>
004012D0  add  esi, ebp
004012D2  mov  ecx, 9F5
004012D7  rep  movs byte ptr es:[edi], byte ptr
```

Figure 9. Fragment of destructive code of W32/NGVCK.dr.gen

4.4 Discussion

The challenge of reverse engineering to understand the real intention of a malware has increased as many techniques implemented by malware authors could thwart the analysis process. In section 4.1, our approach shows that the OEP of a packed executable file can be spotted easily by locating the only incoming nodes. The OEP identification becomes useful when a packed executable implements an anti-debugging method, such as the variations of timing checks. Typical normal step tracing method could not defeat the anti-debugging technique. Therefore, a breakpoint can be set at the OEP to allow the targeted program to only be executed when the execution pointer pauses at the OEP to begin the analysis process.

During the dynamic analysis process, the forward and backward component provides a comprehensive analysis environment for analysts



during the analysis process. For every execution, the particular register and its instructions will be saved in our database. With the backward function to be triggered, the user can travel back to instructions of interest and the particular values of the register will be updated.

The proposed malware analysis via CFG is based on the belief that malware is designed to conduct malicious activities. Therefore, most basic block codes consist of malicious program. As long as the targeted malware program stays in the real body of the executable without any obfuscation technique, the analysts could easily study the malicious code by randomly clicking the nodes generated by the Mini Graph.

5. CONCLUSION

This paper presents the idea and structure of an interpreter of the visualization debugger for both static and dynamic debugger analysis. Our approach enables the monitoring process of distributing tasks and leads to an interactive parallel execution debugging process, where an operator monitors the exploration path of nodes. The proposed static and dynamic de-compilation techniques incorporates visualization graph to provide a comprehensive reverse engineering environment to the security expert in the malware tracing process. With the proposed visualization, the irregularities in binary execution are more intuitively identifiable. Moreover, the backward component increases the efficiency of the reverse engineering process by providing the backtracking capability which enables efficient transitions between execution points in a trace in both forward and backward directions.

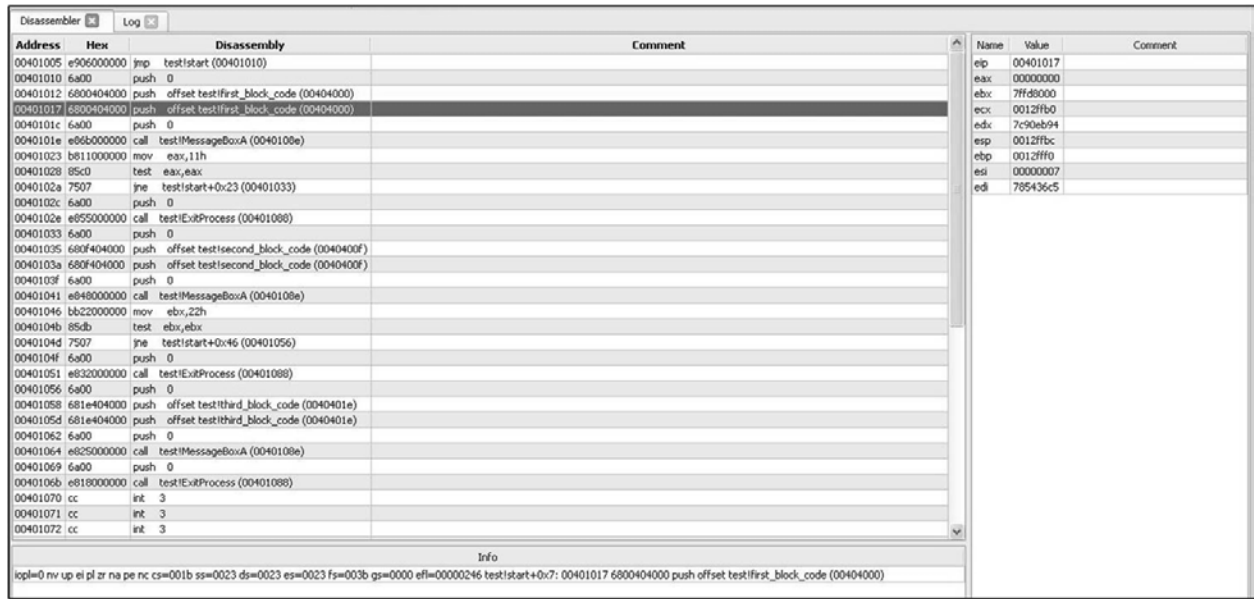
The current version of the visualization debugger is only on ring 3 - Application Level. We plan to enhance our debugger by introducing the ability to debug not only on ring 3 but also on ring 0 – Kernel Level. With the enhancement of the capability to debug up to ring 0, more malware especially kernel related malicious code including rootkit are allowed to perform malicious analysis with our visualization debugger.

REFERENCES:

- [1] Economics, C. 2007 Malware Report: Annual Worldwide Economic Damages from Malware Exceed \$13 Billion. Computer Economics Report, 2007.
- [2] Distler, D. Malware Analysis: An Introduction. 2007.
- [3] Sharif, M., et al., Eureka: A Framework for Enabling Static Malware Analysis, in Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security2008, Springer-Verlag: Málaga, Spain. p. 481-500.
- [4] Eagle, C., The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler 2011 San Francisco: William Pollock. 672.
- [5] Āurfina, L., et al., Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis, in Information Security and Assurance, T.-h. Kim, et al., Editors. 2011, Springer Berlin Heidelberg. p. 72-86.
- [6] Yushuk. Ollydbg debugger and disassembler. Product Description Page. 2000 [cited 2010 20 Jun]; Available from: <http://www.ollydbg.de/>.
- [7] Developers, T.G. GDB: The GNU Project Debugger. Available from: <http://sources.redhat.com/gdb/>.
- [8] Microsoft. Download and Install Debugging Tools for Windows. 2012; Available from: <http://msdn.microsoft.com/en-us/windows/hardware/gg463009.aspx>.
- [9] Russinovich, M. and B. Cogswell. Process Monitor v3.03. 2012; Available from: <http://technet.microsoft.com/en-us/sysinternals/bb896645>.
- [10] Schwarz, B., S. Debray, and G. Andrews, Disassembly of Executable Code Revisited, in Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)2002, IEEE Computer Society. p. 45.
- [11] Cifuentes, C., et al. Preliminary Experiences with the Use of the UQBT Binary Translation Framework. in In Proceedings of the Workshop on Binary Translation. 1999.
- [12] Kruegel, C., et al., Static disassembly of obfuscated binaries, in Proceedings of the 13th conference on USENIX Security Symposium - Volume 132004, USENIX Association: San Diego, CA. p. 18-18.
- [13] Xia, Y., K. Fairbanks, and H. Owen, Visual Analysis of Program Flow Data with Data Propagation, in Visualization for Computer Security, J. Goodall, G. Conti, and K.-L. Ma, Editors. 2008, Springer Berlin Heidelberg. p. 26-35.



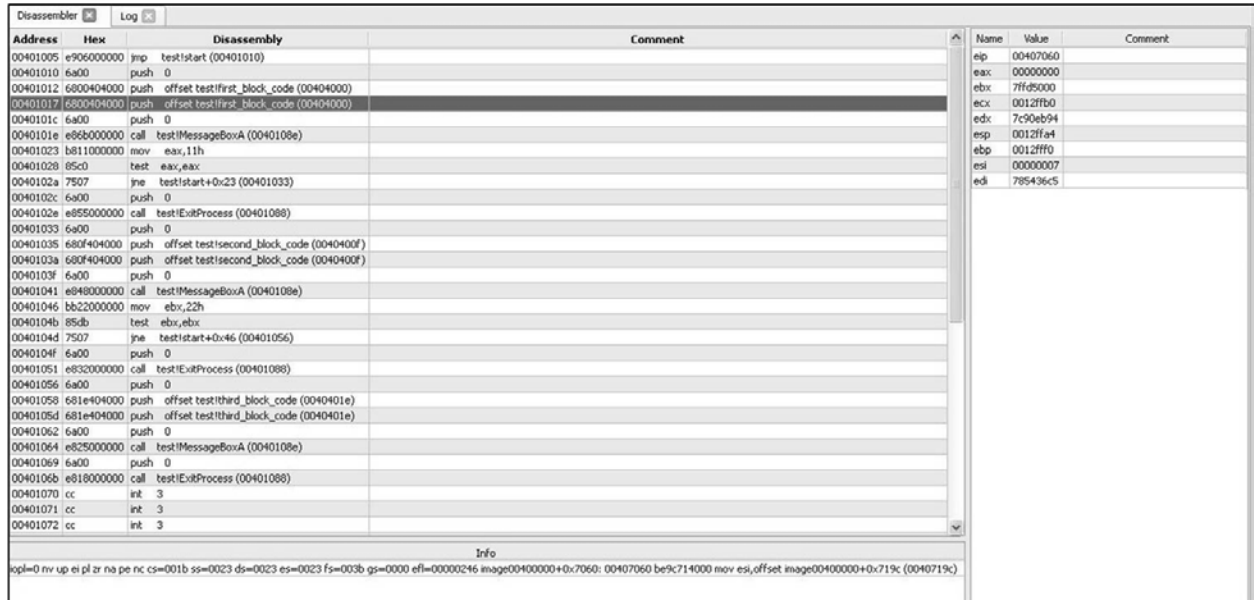
- [14] Madou, M., et al., Hybrid static-dynamic attacks against software protection mechanisms, in Proceedings of the 5th ACM workshop on Digital rights management2005, ACM: Alexandria, VA, USA. p. 75-82.
- [15] Nanda, S., et al., BIRD: binary interpretation using runtime disassembly in Proceedings of the International Symposium on Code Generation and Optimization2006. p. 358-370.
- [16] Quist, D.A. and L.M. Liebrock. Visualizing Compiled Executables for Malware Analysis. in 6th International Workshop on Visualization for Cyber Security 2009. Atlantic City, NJ, USA.
- [17] Wagener, G., R. State, and A. Dulaunoy, Malware behaviour analysis. Journal in Computer Virology, 2008. 4(4): p. 279-287.
- [18] vmware. Choosing a Business Infrastructure Virtualization Solution. Available from: <http://www.vmware.com/virtualization/advantages/>.
- [19] Summerfield, M., Rapid gui programming with python and qt: the definitive guide to pyqt programming2007: Prentice Hall Press. 648.
- [20] Maebe, J. and K.D. Bosschere. Instrumenting self-modifying code. in proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003). 2003. Ghent, Belgium.
- [21] uDraw(Graph). uDraw(Graph) - The powerful solution for graph visualization. 2005; Available from: <http://www.informatik.uni-bremen.de/uDrawGraph/en/index.html>.
- [22] Roundy, K.A. and B.P. Miller, Hybrid analysis and control of malware, in Proceedings of the 13th international conference on Recent advances in intrusion detection2010, Springer-Verlag: Ottawa, Ontario, Canada. p. 317-338.
- [23] Szor, P., The Art of Computer Virus Research and Defense2005: Addison-Wesley Professional.



Address	Hex	Disassembly	Comment
00401005	e9060000	jmp test!start (00401010)	
00401010	6a00	push 0	
00401012	68004000	push offset testfirst_block_code (00404000)	
00401017	68004000	push offset testfirst_block_code (00404000)	
0040101c	6a00	push 0	
0040101e	e86b0000	call test!MessageBoxA (0040108e)	
00401023	b8110000	mov eax,11h	
00401028	85c0	test eax,eax	
0040102a	7507	jne test!start+0x23 (00401033)	
0040102c	6a00	push 0	
0040102e	e8550000	call test!ExitProcess (00401088)	
00401033	6a00	push 0	
00401035	68f40400	push offset test!second_block_code (0040400f)	
0040103a	68f40400	push offset test!second_block_code (0040400f)	
0040103f	6a00	push 0	
00401041	e8480000	call test!MessageBoxA (0040108e)	
00401046	bb220000	mov ebx,22h	
0040104b	85db	test ebx,ebx	
0040104d	7507	jne test!start+0x46 (00401056)	
0040104f	6a00	push 0	
00401051	e8320000	call test!ExitProcess (00401088)	
00401056	6a00	push 0	
00401058	681e4000	push offset test!third_block_code (0040401e)	
0040105d	681e4000	push offset test!third_block_code (0040401e)	
00401062	6a00	push 0	
00401064	e8250000	call test!MessageBoxA (0040108e)	
00401069	6a00	push 0	
0040106b	e8180000	call test!ExitProcess (00401088)	
00401070	cc	int 3	
00401071	cc	int 3	
00401072	cc	int 3	

Name	Value	Comment
eip	00401017	
eax	00000000	
ebx	7ff45000	
ecx	0012ff60	
edx	7c90eb94	
esp	0012ffbc	
ebp	0012fff0	
esi	00000007	
edi	785436c5	

Info
 iopl=0 nrv up ei pl zr na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246 test!start+0x7: 00401017 68004000 push offset testfirst_block_code (00404000)

Address	Hex	Disassembly	Comment
00401005	e9060000	jmp test!start (00401010)	
00401010	6a00	push 0	
00401012	68004000	push offset testfirst_block_code (00404000)	
00401017	68004000	push offset testfirst_block_code (00404000)	
0040101c	6a00	push 0	
0040101e	e86b0000	call test!MessageBoxA (0040108e)	
00401023	b8110000	mov eax,11h	
00401028	85c0	test eax,eax	
0040102a	7507	jne test!start+0x23 (00401033)	
0040102c	6a00	push 0	
0040102e	e8550000	call test!ExitProcess (00401088)	
00401033	6a00	push 0	
00401035	68f40400	push offset test!second_block_code (0040400f)	
0040103a	68f40400	push offset test!second_block_code (0040400f)	
0040103f	6a00	push 0	
00401041	e8480000	call test!MessageBoxA (0040108e)	
00401046	bb220000	mov ebx,22h	
0040104b	85db	test ebx,ebx	
0040104d	7507	jne test!start+0x46 (00401056)	
0040104f	6a00	push 0	
00401051	e8320000	call test!ExitProcess (00401088)	
00401056	6a00	push 0	
00401058	681e4000	push offset test!third_block_code (0040401e)	
0040105d	681e4000	push offset test!third_block_code (0040401e)	
00401062	6a00	push 0	
00401064	e8250000	call test!MessageBoxA (0040108e)	
00401069	6a00	push 0	
0040106b	e8180000	call test!ExitProcess (00401088)	
00401070	cc	int 3	
00401071	cc	int 3	
00401072	cc	int 3	

Name	Value	Comment
eip	00407060	
eax	00000000	
ebx	7ff45000	
ecx	0012ff60	
edx	7c90eb94	
esp	0012ffa4	
ebp	0012fff0	
esi	00000007	
edi	785436c5	

Info
 iopl=0 nrv up ei pl zr na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246 image00400000+0x7060: 00407060 be9c714000 mov esi,offset image00400000+0x719c (0040719c)

Figure 8: Dynamic Analysis via Forward and Backward Tracing