STUDENT SUMMER INTERNSHIP TECHNICAL REPORT

Using a 64-bit Disassembler to Employ Heuristic Analysis of Executable Programs using Hyperion

DOE-FIU SCIENCE & TECHNOLOGY
WORKFORCE DEVELOPMENT PROGRAM

Date submitted:

September 30, 2015

Principal Investigators:

Andrew De La Rosa, DOE Fellow
Florida International University

Joseph Trien, Mentor
Oak Ridge National Laboratory

Florida International University Program Director:

Leonel Lagos Ph.D., PMP®

Submitted to:

U.S. Department of Energy
Office of Environmental Management
Under Cooperative Agreement # DE-EM0000598

FIU | Applied Research Center
FLORIDA INTERNATIONAL UNIVERSITY

ABSTRACT

Hyperion is software that is used to reverse engineer executable binary code to return the program’s functional behavior. Today, the use of behavioral analysis has shown to be a proven and effective way to scan for malware, alongside heuristic analysis, which has been the ‘de-facto’ model of analyzing computers for malware for the past 20 years. But as technology has improved and new systems are using 64 bit programs, so are many malware authors; this requires Hyperion to be updated to meet these new threats and computer attacks.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

The Hyperion Project is an ongoing 10 year project, with the ability to analyze software using mathematical computations. Analyzing the functional behavior of a program is a potentially more accurate way to detect potential malicious activity, since a program may not exhibit malicious behavior during testing. Currently, the security policies we have of standardization, training security personnel, and risk management are good implementations but are not enough to tackle the threat. Hyperion analyzes the structure of a program for all its intended or unintended purposes; by operating on the program semantics and the binaries on the low-level of the program's calling structure, we obtain a more accurate analysis of the overall code of the program. Semantic analysis is guided using structure and behavior computation that in turn use mathematical precision to ensure that all outputs are traceable back to the original input and path taken. The success of this project can be exported for customizable use as well as malware detection and rigorous software development.

Currently, Hyperion can process 32-bit binaries, but there is increasing need for a 64-bit model of Hyperion. In order to proceed with this new model and development, a restructured version of the application is required. Modern compilation and design is one of the fundamental key factors that takes place in this process and requires a new approach, different from the market and current views on tackling malware analysis. Malware analysis today depends on heuristic analysis, which determines if a file is malware depending on if the hash taken from a file matches a database catalog full of malware signatures. While this technique has been in place for several years by many anti-virus companies, this model is slowly becoming obsolete; this is due to many recent efforts by malware authors to hide their files using legitimate programs to execute them (which in turn nullifies this procedure).

Behavioral analysis is a far more precise method of analyzing malware and determining its effects; malware programs today have timing and structured methods that prevent detection and can actually alter data in order to make other files and programs seem malicious. Behavioral methods require more resources because they take into account all of the components that make up the program (including but not limited to timing delays, obfuscation, and third-party packaging).

In order to implement a 64-bit model of Hyperion, we have to take the following into account: 1) The current model of Hyperion has a limit on the size of the file that it could take; 2) Hyperion is built on multiple coding languages (C, C++, Python, etc.), so there are additional files that perform conversions in order for the scripts to understand each other; and 3) Hyperion is set to inspect only .exe executables, and does not have the framework to examine other file types.

2. EXECUTIVE SUMMARY

This research work has been supported by the DOE-FIU Science & Technology Workforce Initiative, an innovative program developed by the US Department of Energy's Environmental Management (DOE-EM) and Florida International University's Applied Research Center (FIU-ARC). During the summer of 2015, a DOE Fellow intern (Andrew De La Rosa) spent 10 weeks doing a summer internship at Oak Ridge National Laboratory, Division of Cyber and Computational Sciences, under the supervision and guidance of Joseph Trien, Director of the Cyber Sciences Division. The intern's project was initiated on June 1, 2015, and continued through August 7, 2015 with the following objectives: 1) Access the GitLab website and read all documentation related to the project; 2) Ensure that all scripts and programs are compiling correctly, with minimal timing issues; 3) Research how to use 64-bit programs in the input structure; 4) Compile the program in C or C++ with the current program structure; and 5) Develop a final paper based on changes made due to 64-bit architecture.

3. RESEARCH DESCRIPTION

A. The ‘H’-Chart Algorithm

The H-Chart is an “algorithm [that] improves upon the simple syntactic construction of a flow chart by only looking at individual machine code instructions.” [4] The following diagram below shows the functional model architecture for analyzing malware behavior.

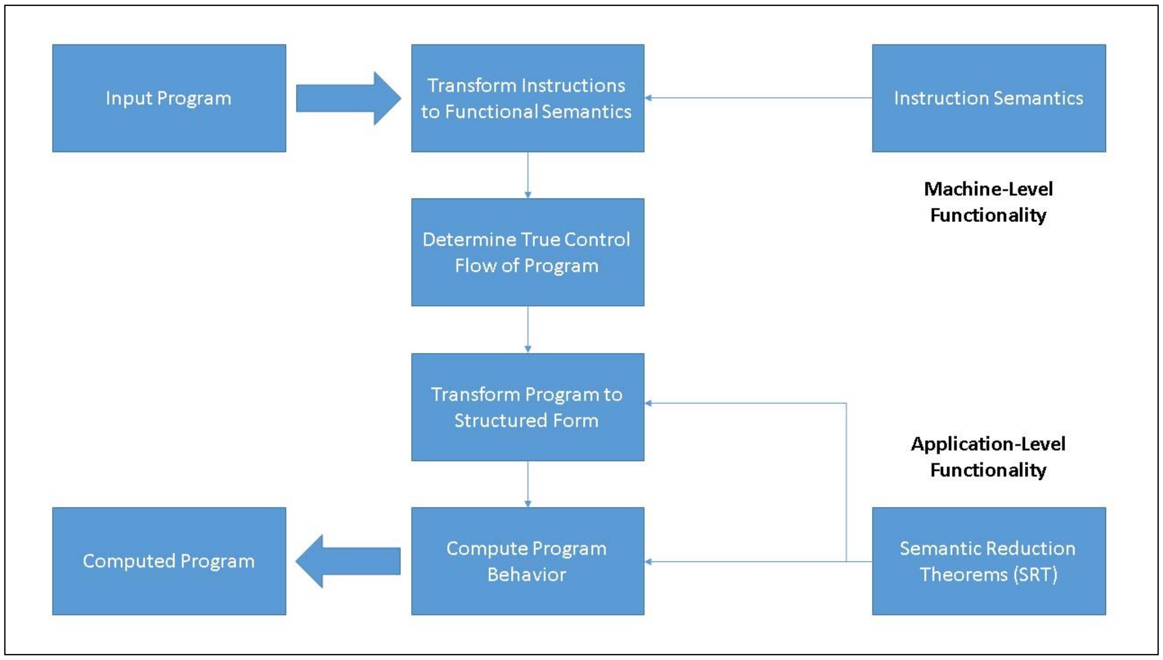

Input Program
Transform Instructions to Functional Semantics
Instruction Semantics
Machine-Level Functionality
Determine True Control Flow of Program
Transform Program to Structured Form
Application-Level Functionality
Computed Program
Compute Program Behavior
Semantic Reduction Theorems (SRT)

Figure 1. Behavior computation system architecture.

The first step in the architecture is to insert a program into the analysis; for testing purposes, Windows executables are used (i.e., .exe). Once the program has been unpackaged, the instructions are transformed into functional semantics and returned. The instructions refer to the assembly language found in any operating system that denotes the human readable code that is used by the computer. In order to correctly deduce the correct instructions, the semantics for instructions are adapted at the machine-level, since the machine decodes the compiled program and executes.

The following step is used to determine the control flow of the program. The true form of a program is created using structured programming, which includes a form of programming as well as drawing a flow control of how a program runs. Programming can be done in a number of ways, but when the compiler has to interpret the high-level language into machine code, a number of problems can occur, such as errors in breakpoints and calling unneeded subroutines that waste resources. For this reason, flow charts help minimize the number of potential redundancies that can occur in loops and timing delays. In some cases, some iterated loops can then be reduced to functions that can help create a linear path instead of loops that can be more difficult to analyze. Figure 2 below shows the basic looping flow control.

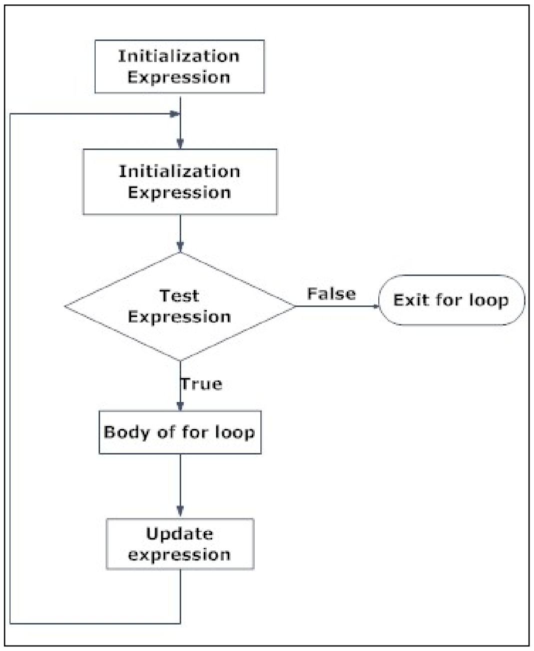


Figure 2. Example for computing flow control.

The next step is to determine the structured form for the program. Structured programming is similar to the 'C' programming with the restriction of not using the 'goto' statement; however, any time that expressions are compared or are used in a loop there are semantics that are introduced to show the end of those sections. Structured programming is done using PDL, which stands for 'Program Design Language'. It is a method for designing and documenting methods and procedures in software. It is related to pseudocode, but unlike pseudocode, it is written in plain language without any terms that could suggest the use of any programming language or library. This language is a very-high level language that supports other high-level languages such as Fortran, Ada, C, and Pascal. The simplicity of PDLs is that they output a written format of the executable code. Figure 3 below shows an example of a statement that is iterated after a 'for' loop.

```
...
for
i:ε 1 to 20 by 2
do
j := table(i) + table (i+1)
print j
od
…
```

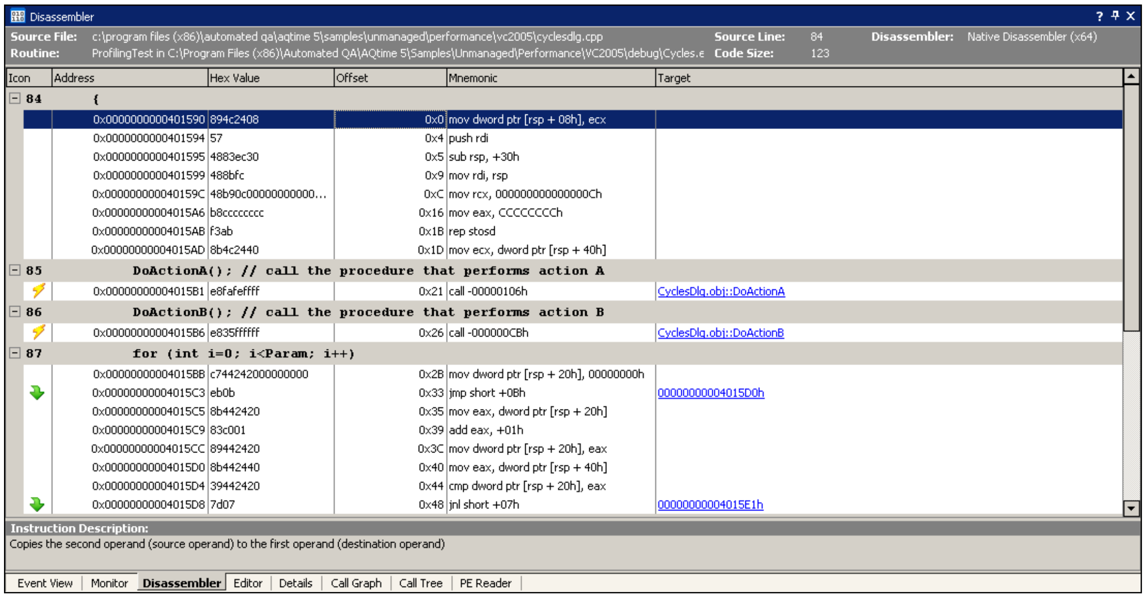
Figure 3. Example of a structured program using PDL.

For each statement that needs to be executed, there must be a 'do' syntax that represents the section of the code that needs to be executed. This is how an entire program is structured; however, depending on the size of the files and libraries, this file can become increasingly large, almost to the point where the exported program is incomprehensible. Instead, this is where semantic reduction theorems (or SRT, for short) come into play. Semantic reduction theorems help reduce the number of iterations that occur, as well as simplifying some functions and calls into an assembly modular code. A simple example is when a piece of code causes a loop to be called every time that a check is done for a specific string in a certain location. This can be reduced to a function call that can be stored as a memory location instead of continually adding these statements to memory that can fill the stack up, rather than leaving space for other functions to process and compute.

Once this step is complete, the program will be completely dissected, which is done by computing the behavior. In order to compute the behavior, the code needs to recompile and execute but the processes do not take place. If the processes execute in the same order and produce the same targets, then the output would match the input and the source code would be deemed the original code of the program. You can configure for which output language (most executables are written in either C or C++). This is completed at the application level because authors can insert different semantic algorithms to reduce the instructions using different scripts.

B. The Disassembler

A disassembler is a program that functions in the opposite manner of an assembler; it takes in a specified program and returns the assembly language from the machine code. Malware today can check if it is really in the computer or is being emulated through third-party software such as VMware or Virtual Box. A disassembler is a powerful tool that attempts to recreate the assembly code from the binary machine code.



Source File: c:\program files (x86)\automated qa\aqtime 5\samples\unmanaged\performance\vc2005\cyclesdlg.cpp **Source Line:** 84 **Disassembler:** Native Disassembler (x64)
Routine: ProfilingTest in C:\Program Files (x86)\Automated QA\AQtime 5\Samples\Unmanaged\Performance\VC2005\debug\Cycles.e **Code Size:** 123

Icon	Address	Hex Value	Offset	Mnemonic	Target
84	**{**				
	0x0000000000401590	894c2408	0x0	mov dword ptr [rsp + 08h], ecx	
	0x0000000000401594	57	0x4	push rdi	
	0x0000000000401595	4883ec30	0x5	sub rsp, +30h	
	0x0000000000401599	488bfc	0x9	mov rdi, rsp	
	0x000000000040159C	48b90c00000000000…	0xC	mov rcx, 000000000000000Ch	
	0x00000000004015A6	b8cccccccc	0x16	mov eax, CCCCCCCCh	
	0x00000000004015AB	f3ab	0x1B	rep stosd	
	0x00000000004015AD	8b4c2440	0x1D	mov ecx, dword ptr [rsp + 40h]	
85	**DoActionA(); // call the procedure that performs action A**				
⚡	0x00000000004015B1	e8fafeffff	0x21	call -00000106h	CyclesDlg.obj::DoActionA
86	**DoActionB(); // call the procedure that performs action B**				
⚡	0x00000000004015B6	e835ffffff	0x26	call -000000CBh	CyclesDlg.obj::DoActionB
87	**for (int i=0; i<Param; i++)**				
	0x00000000004015BB	c744242000000000	0x2B	mov dword ptr [rsp + 20h], 00000000h	
⬇	0x00000000004015C3	eb0b	0x33	jmp short +0Bh	00000000004015D0h
	0x00000000004015C5	8b442420	0x35	mov eax, dword ptr [rsp + 20h]	
	0x00000000004015C9	83c001	0x39	add eax, +01h	
	0x00000000004015CC	89442420	0x3C	mov dword ptr [rsp + 20h], eax	
	0x00000000004015D0	8b442440	0x40	mov eax, dword ptr [rsp + 40h]	
	0x00000000004015D4	39442420	0x44	cmp dword ptr [rsp + 20h], eax	
⬇	0x00000000004015D8	7d07	0x48	jnl short +07h	00000000004015E1h

Instruction Description:
Copies the second operand (source operand) to the first operand (destination operand)

Figure 4. Example of a disassembler extracting machine code into assembly code.

One of the issues that occurs with disassembly is that the method is not bidirectional, so it is not guaranteed that the assembly code will be determined as the absolute replication from the binary. In turn, this occurs to a number of variables that can occur between the source code, the compiler, and the assembler. Often times, there will be a number of jumps that occur and these jumps cause a rift in the code that can sometimes be lost and not written[1].

The disassembler is intelligent enough to recognize the different areas of code such as data, section headers, and the Common Object File Format (COFF) headers as shown below in Figure 5. From 0x00 to 0x17 is the COFF header, which is standard for every file to be identified. This data is the metadata that keeps track of a file's history that can be viewed under the 'Properties' section of a file. The following section is specifically used by the COFF (from 0x18 to 0x33), which is used by the standard COFF Header and identifies key points such as size of the data, size of the file, date and time, etc. for UNIX based systems [2]. The Windows friendly data (from 0x34 to 0x77) provides the same information as COFF does for UNIX, but it is used specifically for the Microsoft Windows version, such as the stack and checks the environment of which Windows is being executed. The directory of the data is the next section and is highlighted as a section preceded called 'data'. In this data, you have several data headers that describe the different types of tables (such as Import Addresses, Resource, Import and Export, and their respective sizes) as well as any global pointers in the data. The final section is the section table that gives the size of the data, and the pointers that are used in execution of the data. These two final sections do not have a definitive size because, unlike the previous sections, the size of these sections depend on the size of the data itself, and not the physical quantitative size which is recorded in the first two sections.

Figure 4 gives a perfect representation of the output of what a disassembler is supposed to do; it returns the assembly code for the inputted executable, with a table for any strings that are able to be extracted. Using these strings, the user is able to navigate through the program and see where different function calls are pointing to as well as see what the final output of a certain process entails. It also helps to know some basic computations in assembly such as: mov, cmp, xor, push, jnl. In assembly, you also have different instances such as 'dword', which is the instantiation of a temporary variable (in high-level languages, you declare a type such as 'int' for integer, and 'char' for character).

[1] These errors often occur when code is written but never used, such as creating a function that when it is called, neither returns nor modifies any values. These breakpoints in the machine code never resolve and usually stop in the graphical representation illogically.

[2] When UNIX became an increasingly popular OS, vendors adapted their file extensions to be able to be used by both Microsoft Windows and UNIX. But in order to accommodate, certain elements in the file structure needed to be added in order for the OS to recognize, for example, text files (.txt)

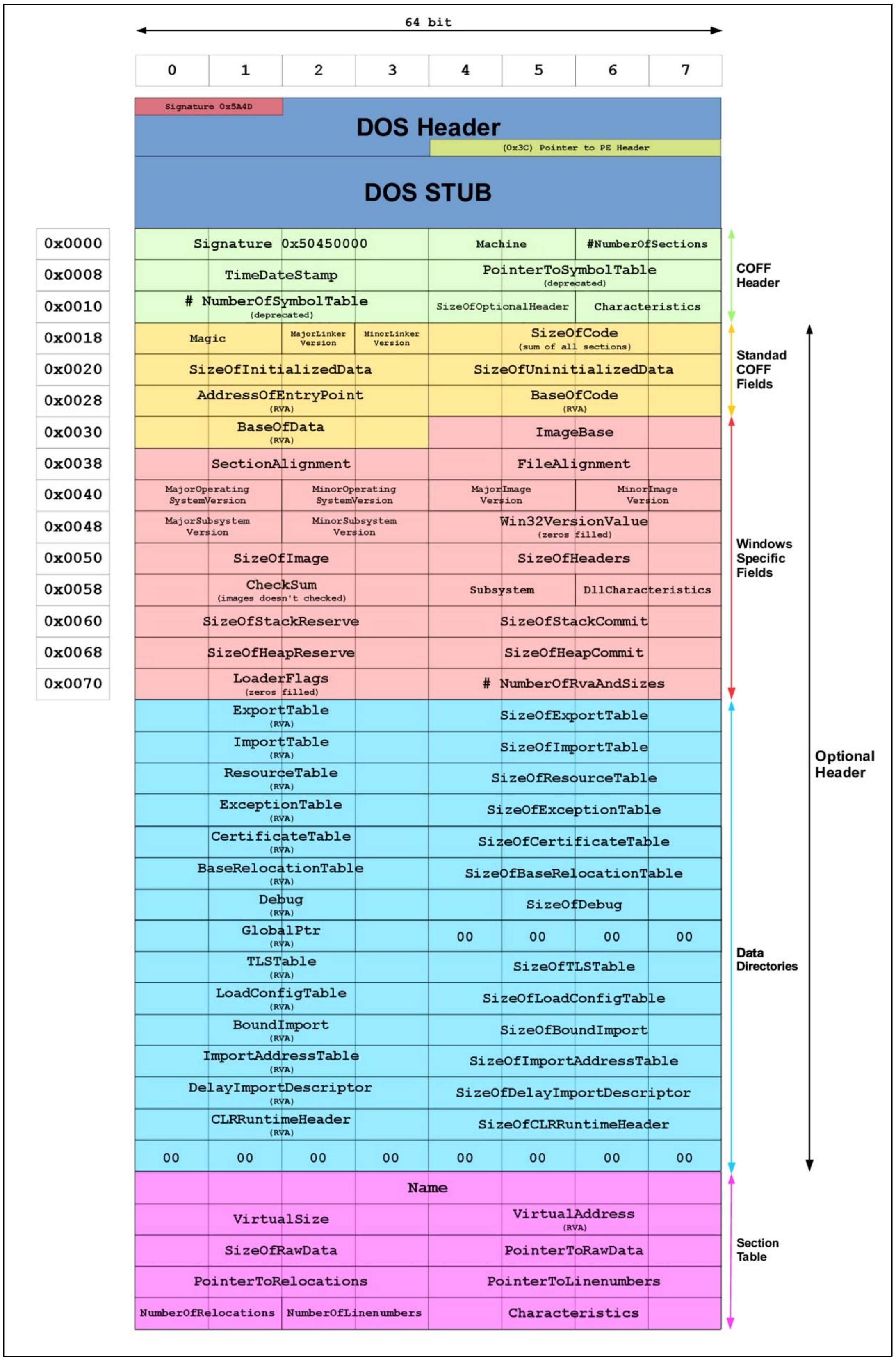


Figure 5. Breakdown of 32-bit portable executable file.

4. RESULTS AND ANALYSIS

The 32-bit code compiler will sometimes generate code because the stack size will not change in the middle of processing the function – unlike 64-bit instructions, the stack is always being used – which is why the code is never generated. This is due to the functionality of the push and pop instructions; in 32-bit, the push and pop can change the size of the stack accordingly, so the size of the stack can vary, depending on how many functions are called and how much data is being moved and processed. In a 64-bit, the push and pop functions do not operate the same because as the size of the stack increases, it never can be reduced until the stack is cleared at the end. In short, the size of the stack in 32-bit is variable, while in 64-bit it is fixed, which makes it harder to see the functionality of a program. There are some instances that it is not clear the number of parameters being passed, so the only option would then be to analyze the strings for clarification.

One method I have used to analyze coded structure is to add breakpoints and see the changes in memory (this is what I used when I was analyzing malware that changed the registry and inserted itself in runtime). The idea would be the following: Hyperion loads the executable and extracts the assembly code in order to show the registers and data being moved and/or accessed – and the output would be a programmed code that is outputted; in 64-bit programs, this is not practicable because the stack is constantly being used in order to structure the data and allocate the necessary memory to move the data.

There are some advantages in using 64-bit. While most programs in 64-bit use pointers and registers affiliated with 64-bit, there is sometimes the occasional 32-bit instruction that is placed in. For example, most integers that are used are 32-bit, because they only require 4 bytes to store the numerical value; it is costly and a waste of time to call these values as a double or a float when the amount of available memory is not enough. During these runtimes of 32-bit instructions, we can see more precisely what is being accessed and moved.

In 32-bit, the most common registers being accessed are the registers 0-7, which are: EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP. When 64-bit was released, the registers remained the same, except instead of an ‘E’ there is an ‘R’ in front (i.e.: EAX = 32-bit, RAX = 64-bit). This also led to an expansion of more utilized registers, R8-R15. If any lower bit architecture wanted access to these additional registers, they would add an additional letter to them (i.e.: R8 = 64-bit, R8D = 32-bit, R8W = 16bit, R8B = 8bit). Most 32-bit programs do not have the need for the additional R8-R15 registers but they are available if necessary. The use of these registers is also an indication that it is being run in a 64-bit environment.

In researching the 64-bit disassembler, it does exist for the ARM processor; however, it is not entirely compatible. The ARM processor is built for x86/64 compatibility unlike Microsoft Windows that separated 32 and 64-bit. What ARM did was that it tried to use the resources for 32-bit and include 64-bit when necessary. When Windows 7 was introduced, many users were still used to programs running in 32-bit mode, but when programs transitioned over to 64-bit, a compatibility mode was added to allow for stability and continued use of these programs. These resources are separated into 2 separate folders: 1) System (for 64-bit programs), and 2) System32 (for 32-bit programs). There are many instances today where 64-bit programs can use the resources from System32 because of the versatility of some applications. The negative aspect of Microsoft Windows is that it needs to have a compatible copy of each 64-bit file in the event that a 32-bit program needs access to a specific resource.

Another issue that is coming across a 64-bit disassembler is extracting information from a 64-bit register; in order to extract data from a 32-bit program, there is a push and pull request that configures the data to be moved to and from the stack. This action causes the stack to release information because the size of the stack is variable and can pick up on the data that is being released while it is running. In 64-bit mode, this is not as efficient; the stack is not variable, but rather fixed. So while there are commands that push and pull the data, the stack is not cleared in the event of a process error and the event can be duplicated without causing massive damage to the system. However, because not all the registers that apply to the 64-bit mode behave in the same manner, it can be difficult to decipher some binary code in the assembly, even something as simple as adding two numbers (for example, in the event that the result of the integer passes the limit of 32-bit – which would then be considered an overflow).

5. CONCLUSION

The 32-bit model for Hyperion has been completed and the only requirements that are needed are maintenance. To improve Hyperion and expand it to 64-bit mode, several improvements need to be implemented. First, research on extracting information from a 64-bit stack from Intel cards needs to be done, because most computers use either AMD or Intel architecture. The next step is to improve the ‘H’-Chart algorithm and the disassembler to allow manual configurations since they only perform for 32-bit and need more instructions for the R8-R15 registers. The final step is to implement a graphical user interface to allow the program to be used more effectively since it is only available as an executable package from the terminal. Hyperion is a model for the future when it comes to reverse engineering malware attacks.

6. REFERENCES

[1] D. Kusswurm, Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX, New York City, New York: Springer Science, 2014.

[2] M. Sikorski and A. Honig, Practical Malware Analysis, San Francisco, California: No Starch Press, 2012.

[3] D. Quist, L. Liebrock and J. Neil, "Improving Antivirus Accuracy with Hypervisor Assisted Analysis," *Journal in Computer Virology,* vol. 7, no. 2, pp. 121-131, May 2011.

[4] R. Linger, T. Daly and M. Pleszkoch, "Function Extraction (FX) Research for Computation of Software Behavior: 2010 Development and Application of Semantic Reduction Theorems for Behavior Analysis," Software Engineering Institute, Hanscom AFB, 2011.