

An Approach towards Disassembly of Malicious Binary Executables

A Thesis

Presented to the

Graduate Faculty of the

University of Louisiana at Lafayette

In Partial Fulfillment of the

Requirements for the Degree

Master of Science

Aditya Kapoor

Fall 2004

© Aditya Kapoor

2004

All Rights Reserved

An Approach towards Disassembly of Malicious Binary Executables

Aditya Kapoor

APPROVED:

Arun Lakhotia, Chair
Associate Professor of Computer Science

William R. Edwards
Associate Professor of Computer Science

Dmitri Perkins
Assistant Professor of Computer Science

C. E. Palmer
Dean of the Graduate School

To Mom and Dad

Acknowledgements

I thank my advisor, Dr. Arun Lakhotia for his valuable guidance, extraordinary support, inspiration, and encouragement. This thesis would never have been conceptualized without the ideas and motivation that he provided me. I greatly appreciate his patience and the trust he showed in me throughout my thesis. He was always there to support me both morally and academically whenever I was in a fix. My gratitude for him cannot be expressed in a paragraph.

I thank my parents and my sister for their never-ending encouragement, trust, and support during my lows and my highs in the research period. I am grateful to Dr. Andrew Walenstein for reviewing my thesis document in a short time and giving me helpful insights. Thanks to Eric Uday, Prashant Pathak, Nitin Jyoti, and Firas Bouz who gave me helpful feedback during all times in my thesis and shared intriguing discussions about the challenges I faced.

Finally, I would also like to thank my roommates Bunty Agrawal, Alok S. Singh, and Jai Dialani for their constant words of support and encouragement in the last two years and the good food they cooked for me.

TABLE OF CONTENTS

1	INTRODUCTION	1
1.1	MOTIVATION.....	1
1.2	RESEARCH OBJECTIVES	4
1.3	RESEARCH CONTRIBUTIONS.....	4
1.4	IMPACT OF THE RESEARCH.....	4
1.5	ORGANIZATION OF THESIS.....	5
2	BACKGROUND	6
2.1	DISASSEMBLY OVERVIEW.....	6
2.2	DISASSEMBLY CHALLENGES.....	7
2.3	METHODOLOGIES OF STATIC DISASSEMBLY	9
2.3.1	<i>Flow Insensitive Analysis</i>	9
2.3.2	<i>Flow Sensitive Analysis</i>	10
2.3.3	<i>Combined Linear Sweep and Recursive Traversal</i>	12
2.3.4	<i>Interactive Disassembly</i>	12
2.3.5	<i>Disassembly Heuristics</i>	13
3	PE ARCHITECTURE AND INTEL INSTRUCTION FORMAT.....	14
3.1	PE FILE FORMAT	14
3.2	INTEL INSTRUCTION FORMAT	15
3.3	INTEL OPCODE MAPS.....	18
3.4	ADDRESSING MODES	18
4	SEGMENTATION OVERVIEW.....	19
4.1	SEGMENTATION EFFICACY.....	19
4.2	SEGMENT STRUCTURE	21
4.3	SEGMENT RELATIONSHIPS	21
4.4	SEGMENT CONSTRUCTION	22
4.5	EXPLANATION OF ALGORITHM WAYPOINTS.....	27
4.6	SEGMENT OVERLAPS AND SPECIAL CASES	29
4.7	ILLUSTRATIVE EXAMPLE	31
5	SEGMENT PRUNING VIA SEGMENT CHAINING.....	33
5.1	SEGMENT-CHAINING.....	33
5.2	ALGORITHM: CHAINING AND PRUNING.....	34
5.3	EXAMPLE: SEGMENT CHAINING AND PRUNING.....	37
6	EVALUATION AND RESULTS	38
6.1	THEORETICAL LIMITATIONS AND ASSUMPTIONS	38
6.1.1	<i>Attacking the Assumptions</i>	38
6.1.2	<i>Attacking the Implementation</i>	40
6.2	EXPERIMENTAL SETUP AND ANALYSIS	41
6.2.1	<i>Setup</i>	41

6.2.2	<i>Analysis Parameters</i>	42
6.2.3	<i>Data Analysis: "Clean" Executable</i>	44
6.2.4	<i>Data Analysis: "Malicious" Executable</i>	47
6.2.5	<i>Data Analysis: Non-Executables</i>	49
6.2.6	<i>Inference of Data Analysis</i>	52
7	RELATED WORK	54
8	CONCLUSION AND FUTURE WORK	58
	BIBLIOGRAPHY	60
	ABSTRACT	62
	BIOGRAPHICAL SKETCH	63

LIST OF FIGURES

FIG. 1-1. STAGES IN STATIC ANALYSIS OF BINARY [15].....	2
FIG. 1-2. OBFUSCATION THROUGH JUNK INSERTION.	3
FIG. 2-1. AN EXAMPLE SHOWING DISASSEMBLY DRAWBACKS USING LINEAR SWEEP	9
FIG. 2-2. OBFUSCATION BY JUNK BYTE INSERTION (BEAGLE.H).....	10
FIG. 2-3. COMPARISON OF TARGET RESULTS OF DIFFERENT DISASSEMBLY TECHNIQUES...	12
FIG. 3-1. FILE FORMAT OF A PE FILE.	15
FIG. 3-2. INTEL INSTRUCTION FORMAT	16
FIG. 3-3. ARCHITECTURE OF MOD R/M BYTE AND SIB BYTE.....	17
FIG. 3-4. A 32 BIT INSTRUCTION AND ITS STRUCTURE	17
FIG. 4-1. EXAMPLE SHOWING CONTROL FLOW OF PROGRAM STARTING AT BYTE 0	20
FIG. 4-2. EXAMPLE SHOWING CONTROL FLOW OF PROGRAM STARTING AT BYTE 1	20
FIG. 4-3. STRUCTURE OF A SEGMENT	21
FIG. 4-4. STEPS OF SEGMENT CREATION.....	23
FIG. 4-5. SEGMENT TERMINATION	24
FIG. 4-6. PSEUDOCODE FOR SEGMENTATION ALGORITHM, LEFTMOST COLUMN SPECIFIES CODE WAYPOINTS	26
FIG. 4-7. MAIN PROGRAM PSEUDOCODE FOR COMPUTING SEGMENTS	27
FIG. 4-8. ILLUSTRATIVE EXAMPLE OF SEGMENT OVERLAPS	30
FIG. 4-9. EXAMPLE SHOWING SPECIAL CASE OF OVERLAPPING	30
FIG. 4-10. EXAMPLE 2: EXCERPT OF ASSEMBLY CODE FROM A SAMPLE PROGRAM.....	31
FIG. 4-11. EFFECT OF APPLYING SEGMENTATION ALGORITHM ON GIVEN CODE BLOCK, GREY AREA SHOWS END OF SEGMENT.	32
FIG. 5-1. VARIOUS STAGES OF SEGMENTATION.....	34
FIG. 5-2. PSEUDOCODE FOR SEGMENT-CHAINING ALGORITHM, LEFTMOST COLUMN SPECIFIES CODE WAYPOINTS.....	36
FIG. 5-3. PSEUDOCODE FOR DELETING A SEGMENT	37
FIG. 5-4. EXAMPLE SHOWING SEGMENT PRUNING BY CHAINING	37
FIG. 6-1. RUNTIME SELF-MODIFYING OBFUSCATION (NETSKY.Z).....	39
FIG. 6-2. OBFUSCATION THROUGH EXCEPTION HANDLING.....	40
FIG. 6-3 PRUNING OF INVALID INSTRUCTIONS: CLEAN EXECUTABLES.....	45

FIG. 6-4 SEGMENT COMPARISONS: CLEAN EXECUTABLES	46
FIG. 6-5. CODE INFLATION INDEX: CLEAN EXECUTABLES	46
FIG. 6-6 PRUNING OF INVALID INSTRUCTIONS: MALICIOUS EXECUTABLES.....	48
FIG. 6-7 SEGMENT COMPARISONS: MALICIOUS EXECUTABLES	48
FIG. 6-8 CODE INFLATION INDEX: MALICIOUS EXECUTABLES	49
FIG. 6-9 SEGMENT COMPARISONS: NON-EXECUTABLES	51
FIG. 6-10 PRUNING OF INVALID INSTRUCTIONS: NON-EXECUTABLES.....	51
FIG. 6-11 CODE INFLATION INDEX: NON-EXECUTABLES	52

LIST OF TABLES

TABLE 6-1. RESULTS: CLEAN EXECUTABLE (INSTRUCTION, SEGMENTS, AND OVERLAPS).	44
TABLE 6-2 RESULTS: MALICIOUS EXECUTABLE (INSTRUCTION, SEGMENTS, AND OVERLAPS)	47
TABLE 6-3 IMAGE FILES USED FOR DATA ANALYSIS.....	49
TABLE 6-4 RESULTS: NON-EXECUTABLE (INSTRUCTION, SEGMENTS, AND OVERLAPS).....	50

1 Introduction

1.1 Motivation

In the recent past computer security has become an issue of foremost importance for individuals, businesses, and governments. Hostile programmers, who write programs with malicious intents of collecting private information, spread spam, etc. breach the existing security measures. Whenever these hostile programmers, specifically virus/worm writers, succeed in spreading a virus or a worm, there is a significant loss to businesses. For example, mi2g website [6] quotes that within one quarter the NetSky worm and all its A - Q variants put together, had already caused between \$35.8 billion and \$43.8 billion of estimated economic damages worldwide. The website also quotes that, in March, combined losses due to the three worms Beagle, MyDoom, and NetSky crossed the \$100 billion mark within a week.

The war between hostile programmers and antivirus writers resembles any classic arms escalation. The first step in countering the malicious attacks is to identify the malicious programs. Antivirus companies use several dynamic and static analysis techniques to identify malware [25]. Most of the Anti-Virus (AV) tools depend upon knowledge of what are called “virus signatures,” which are nothing but patterns of system calls. If these AV tools can find a particular signature in their database of currently known patterns, they raise an alarm. To identify signatures from a malicious program or to understand and counteract the malicious behavior, we need to analyze the executable. This typically requires converting the byte sequence of an executable to an intermediate representation.

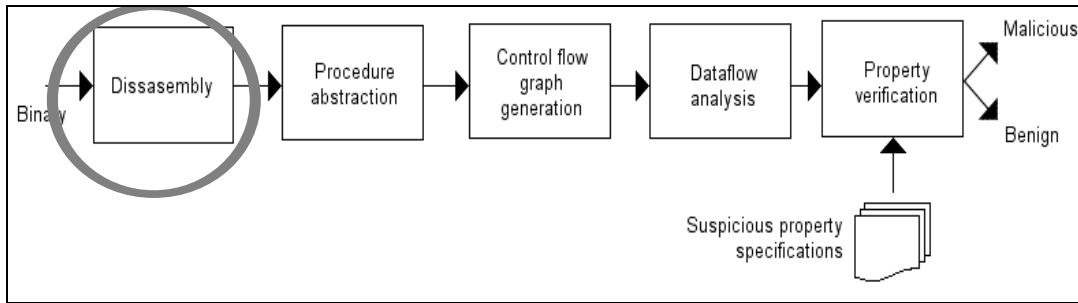


Fig. 1-1. Stages in static analysis of binary [15].

According to [15], the architecture shown in Fig. 1-1 is a proposed staged architecture for binary malware analysis. Disassembly is the first step, and it generally must occur before subsequent analysis can take place. Current disassemblers are used for different purpose, such as binary rewriting for efficiency [20], portability [11, 23] or program maintenance when source code is not available. The problem of disassembly becomes a research issue when the disassemblers are used to analyze malicious codes. The current disassembly algorithms are based on many assumptions such as entry point and control flow information, which are easy targets of malicious code writers.

Techniques used by code-obfuscators [19], for the purpose of protecting intellectual property, can be used by malicious code writers. These writers having malicious intents also rely on many of the existing virus creation tools [5] and obfuscation techniques, such as junk byte insertion, computed jumps, and self-modifying code, to hinder the disassembly process of the current algorithms [14, 17].

There are several examples of virus code and obfuscators that motivate this research. A Few recent examples are self-modifying code of mass mailing worm Netsky.Z and Beagle.H (Junk byte Insertion) that puts the current disassemblers into disarray. Fig. 1-2 shows the disassembled code of Beagle.H obtained from the open

source debugger Ollydbg. Column 1 shows the incorrect disassembly generated from Ollydbg, while column 2 shows the correct disassembly. It is evident from the figure that the virus writer has successfully hidden instructions starting at location 0040A006 and 0040A00A.

Location	Column 1 (Disassembly Ollydbg)		Location		
	Hex	Disassembly			
0040A000	60	PUSHAD	0040A000	60	PUSHAD
0040A001	E8 01000000	CALL 0040A007	0040A001	E8 01000000	CALL 0040A007
...			...		
0040A006	E8 83C404E8	CALL E845648E	0040A006		
0040A007			0040A007		
...			...		
0040A00A			0040A00A		
0040A00B	0100	ADD DWORD PTR DS:[EAX],EAX	0040A00B		

5 bytes

Fig. 1-2. Obfuscation through junk insertion.

The foremost challenge for a disassembly algorithm is to distinguish between code and data, where data can be embedded in code. In the automated analysis of malware the disassembly algorithm should be:

- *Safe*: all the possible instructions should be disassembled such that no code is hidden from the analysis.
- *Precise*: minimum data being incorrectly recognized as code.

The algorithm proposed in this thesis is termed “segmentation algorithm” and it assures safe disassembly. It analyzes each byte for potentially starting an instruction and, thus, finds all the code. Segmentation approach does not leave any code behind from disassembling. For preciseness, it divides the program into segments to remove illegal instructions.

In an environment that is code intensive, there is a high possibility of data being interpreted as code, since out of 512 possible Intel opcodes 398 are valid opcodes. Also, since each byte can start a valid instruction, there is also a need to check on what we call “Code Inflation,” (i.e., how many bytes are present in more than one instruction).

1.2 Research Objectives

The aim of this research is to propose and implement a disassembly method that leaves no code behind from being disassembled, making it harder for malicious code or obfuscated binaries to hide their code, thus raising the bar for hostile programmers.

1.3 Research Contributions

The main contribution of this thesis is a novel approach towards disassembly of malicious and obfuscated binary executables. The main advance comes from a segment-based framework that makes the disassembly process independent of flow analysis and entry point location and, hence, difficult to attack by common obfuscation routines. Instead of the traditional ordered instruction sequence, the output of the disassembler is a collection of potentially overlapping segments, where each segment defines a unique disassembly of some portion of the executable. The technique examines each and every byte of the program, and it treats each byte as a potential instruction. The approach makes a minimum number of assumptions and, hence, is more robust against obfuscation attacks.

1.4 Impact of the Research

The proposed disassembly technique provides a novel disassembly method that can be used to give more precise disassembly in the presence of obfuscations that use indirect jumps, jump in the middle of instruction, self-modifying code, or control flow

obfuscations. The segmentation-based disassembler can be used to augment the current AV analysis systems, or to reverse engineer obfuscated code. The assembly code, generated from the proposed disassembler, can then be fed to subsequent stages of the pipeline shown in Fig. 1-1.

1.5 Organization of Thesis

Chapter 2 provides an overview of disassembly research problems and introduces the main prior disassembly approaches. Intel instruction format and win32 binary executable architecture are explained in chapter 3. The main segmentation algorithm is explained in chapters 4 and 5. Chapter 4 introduces our framework for identifying segments of code and chapter 5 describes heuristics to reduce false positives by pruning segments that lead to data or other corrupt segments. Chapter 6 discusses the evaluation method and results. Chapter 7 presents related work. Future Work and Conclusions are covered in chapter 8, followed by the bibliography.

2 Background

This chapter outlines the process of disassembling binary executables. It also describes the goals and challenges for disassembling malware, along with providing an overview of main prior disassembly techniques.

2.1 Disassembly Overview

Reverse engineering binary executables is a necessary step for rewriting binaries for efficiency [20], or portability [11, 23]. It is also used for maintaining programs where source code is not available, and for detecting malicious programs. Reverse engineering may include recovering the inherent structure and design of program. Techniques for recovering source from binary generally fall into two categories, i.e., either disassembly [14, 17, 22, 24] or decompilation.

Almost all the antivirus groups use disassemblers frequently to analyze the behavior of suspect programs [25]. The algorithms used in the disassemblers, however, are not written with malicious program in mind. This leads to several problems due to code obfuscation such as junk insertion and hiding target of jump instruction [15]. We experienced this first hand, when studying the disassembly of the win32.evol worm, using the leading commercial disassembler IdaPro [2]. The code of win32.evol makes a jump into the middle of a valid instruction, throwing the disassembler and the programmer off the trail of valid instructions. More generally, we may expect that using technologies developed for a “friendly” programmer are unlikely to withstand “attacks” from a “hostile” programmer.

2.2 Disassembly Challenges

Disassembly can be divided into two categories: dynamic disassembly and static disassembly. During dynamic disassembly a program is executed, and during execution, the instructions being executed are traced. A main drawback of this approach is that any given run may trace through only a subset of the possible execution paths, and even after several runs there is no guarantee that all the paths are executed. Take, for example, a worm, which executes the malevolent payload only on certain dates. Static analysis techniques can give more complete results for disassembly because they can analyze the binary as a whole.

A major challenge in correctly disassembling malware is rooted in the von Neumann architecture, where instructions are indistinguishable from data [13]. The problem is perhaps worst for self-modifying code, where code is treated as data, and what was once data becomes executable. A disassembly algorithm could fail, either by incorrectly interpreting some instruction as data (*False Negative*) or by incorrectly interpreting some data as an instruction (*False Positive*). When analyzing malicious code, especially when the analysis is automated, false negatives may lead to an unsafe analysis, i.e., the malicious parts of the code might be overlooked. On the other hand, false positives can also be damaging by overwhelming the engineer or the tool analyzing the binary.

Malware authors intentionally try to defeat static disassembly. They use tricks like jumping into the middle of a valid instruction or use many computed jumps which make it difficult to determine the jump targets statically. Further tricks used to confuse disassemblers include:

- *Junk insertion:* Obfuscators or hostile programmers insert some “junk” bytes between valid instructions to thwart the disassembly. These junk bytes are normally never executed as they are jumped over.
- *Using Exception Handlers to transfer control:* An exception, such as an invalid instruction or referencing a protected memory page, causes a trap or fault, which the operating system catches and transfers control to code called exception handler. This obfuscates jumps, which can cause disassembler to altogether miss the code at the jump target
- *Self-modifying code.* Self-modifying code changes itself dynamically at runtime. Static disassemblers would need to be able to compute the outputs in order to be able to disassemble correctly.
- There are many other novel approaches proposed by [17] such as:
 - *Call conversion:* This obfuscation technique changes the return location of call instruction. The program does not return to the instruction just after call rather it is manipulated to return at a predefined offset from the calling location. The bytes between the offset and location just after call can be filled by junk bytes to confuse disassembler.
 - *Opaque predicates:* In this technique the obfuscators can change all the unconditional *jumps* and *call* to conditional *jumps* and *call*. The branch that is always taken is known. Malicious writers insert junk data on the location of branch that is never taken.
 - *Jump table spoofing:* Artificial insertions of jump tables that are not visited at run time.

2.3 Methodologies of Static Disassembly

Static analysis of a binary is done through methods that are termed either flow sensitive or flow insensitive.

2.3.1 Flow Insensitive Analysis

Flow insensitive algorithms for disassembly do not take control flow of the program in account. In particular they do not use control statements (jumps, calls etc) to affect the choice of which bytes to disassemble. One widely used flow insensitive method for disassembly is the Linear Sweep (LS) algorithm [17, 22-24]. LS starts disassembling instructions from the first executable byte and continue disassembling it until either the program end is reached or it encounters an invalid instruction code. In case LS implementation encounters such an invalid code, it generally either terminates or continues from the next byte.

Consider the sequence of 10 bytes illustrated in Fig. 2-1. In the figure each column represents a byte, and each arrow represents an instruction. Bytes 5 and 6 are junk bytes and are not part of the program code. LS starts disassembling from byte 0 and would return Inst 0, Inst 1 and Inst2 as disassembly result along with bytes 5 and 6 (if bytes 5 or 6 starts an instruction). A hostile programmer can craft bytes 5 and 6 to confuse the disassembler, since now we get four instructions Inst 0, Inst1, Inst 2 and instruction consisting of bytes 5 and 6.

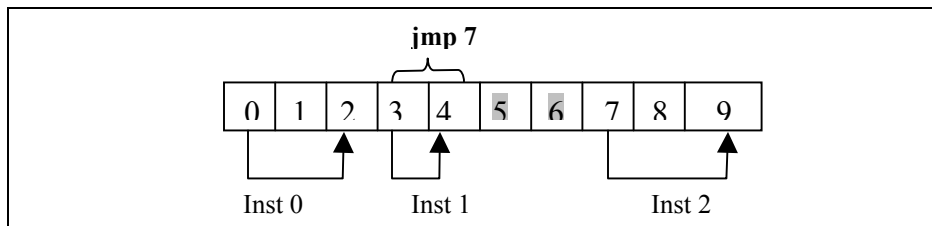


Fig. 2-1. An example showing disassembly drawbacks using Linear Sweep

Fig. 2-1 shows code of a variant of mass mailing worm beagle.h. Column 1 shows the output of the open source debugger Ollydbg, while column 2 shows the desired disassembly. At location 0040A001 the length of the instruction is 5 bytes. LS assumes that the next instruction starts at 0040A006 and disassembles from there, resulting in 5 byte call instruction (*CALL E845648E*), and so the next disassembled instruction starts at 0040A00B which is a junk byte. The junk byte is also a 5-byte call instruction code, which throws off disassembly because the next instruction is supposed to start at 0040A007. Since E8 is an opcode for *call* instruction, it looks like a legitimate instruction starting at 0040A006. The virus writer chose E8 since he has to insert an opcode that starts a valid instruction else LS can raise alarms.

Location	Column 1 (Disassembly Ollydbg)		Column 2 (Actual Disassembly)	
	Hex	Disassembly	Hex	Disassembly
0040A000	60	PUSHAD	60	PUSHAD
0040A001	E8 01000000	CALL 0040A007	E8 01000000	CALL 0040A007
...				
0040A006	E8 83C404E8	CALL E845648E		
0040A007			83C4 04	ADD ESP, 4
...				
0040A00A			E8 01000000	CALL 0040A010
0040A00B	0100	ADD DWORD PTR DS:[EAX],EAX		

Fig. 2-2. Obfuscation by junk byte insertion (Beagle.H)

2.3.2 Flow Sensitive Analysis

Flow sensitive analysis uses control flow information to determine which bytes to disassemble. One of the widely used flow sensitive algorithms is Recursive Traversal (RT) [11, 17, 22-24]. It starts disassembly at the program's main entry point and

disassembles linearly, but as soon as it hits a branch instruction, it jumps to target and again starts disassembling from there. RT disassembles a program, by following all the possible targets of branch instructions encountered. Consider again Fig. 2-2, if we follow RT we get the correct disassembly as shown in column 2. Since at location 0040A001 it will jump to location 0040A007 and starts the disassembly from there. Thus RT ensures unreachable program code at location 0040A006 is not included in the disassembled code.

The main weakness of Recursive Traversal is that it requires precise and accurate control flow information, i.e., it must first precisely identify control flow successors for each control transfer operation in the program. But determining correct control flow is made extremely challenging by indirect jumps, indirect jumps via jump table that does not have bounds check associated with them, and self-modifying code. There are cases when a set of possible targets cannot be statically determined, and hence result in both false positives (in case of overestimation of targets) and false negatives (in case of underestimation of targets). Also RT can be defeated by inserting junk bytes just after a call instruction. Since it assumes that control returns to the next instruction after a call.

Fig. 2-3 illustrates the under and over-estimation of LS and RT, the results of LS and RT are illustrated by inner gray ovals. Overestimation of targets can lead to target locations that are not part of the code and thus can disassemble some data. Underestimation on the other hand might miss target locations where valid code is starting. We say disassembly is precise and safe if we don't have any false negatives and minimum false positives. The dotted oval shows the target of the proposed algorithm.

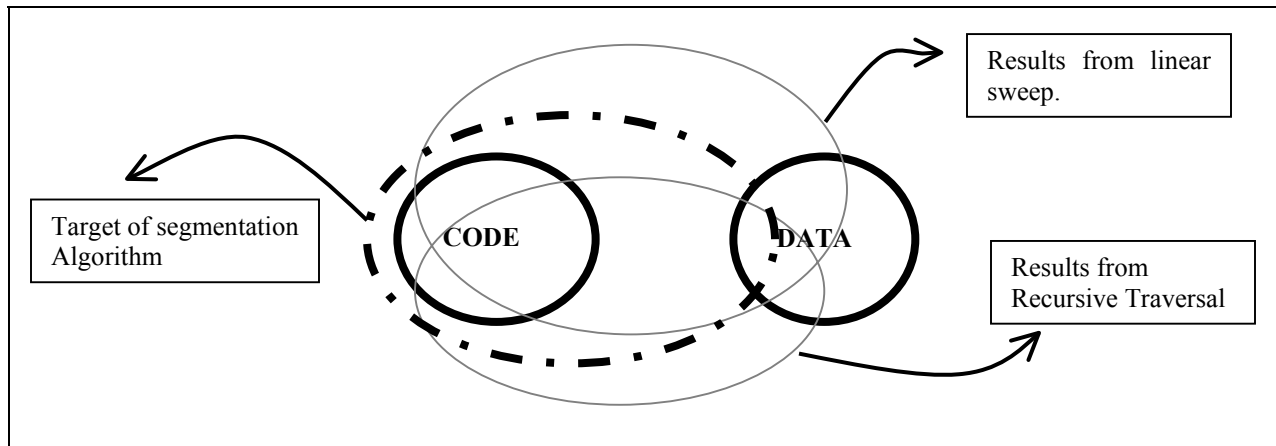


Fig. 2-3. Comparison of target results of different disassembly techniques

2.3.3 Combined Linear Sweep and Recursive Traversal

A hybrid approach that combines both LS and RT is proposed by [22]. The hybrid algorithm disassembles using Linear Sweep and verifies the disassembly a function at a time, using Recursive Traversal. This approach, in some cases, can detect and identify some of incorrect disassembly. This approach also fails in case of call conversion, opaque predicates and self-modifying code.

2.3.4 Interactive Disassembly

Due to drawbacks of current algorithms, some reverse-engineering tools use interactive disassembly techniques [2, 9]. One of the most powerful interactive disassemblers is IDA-Pro [2]. The main advantage with interactive disassembly tools is that the users can choose some bytes that they think might be code and start disassembly from that point onwards. Interactive disassembly is primarily a manual process, but the users are allowed to check validity of their thoughts. Interactive disassembly is laborious and may be error prone due to human intervention.

2.3.5 Disassembly Heuristics

LS and RT- based techniques also have false negatives as well as false positives. To reduce these false negatives heuristics can be used to restart the disassembly in the “*gaps*” of untouched bytes. One such heuristic is termed “Speculative Disassembly” [11]. In this technique, the undisassembled portions of the code are made targets of indirect jumps and a “speculative bit” attached to the target is set. The disassembly of a particular code area is stopped if an invalid instruction is encountered. The disassembly resumes from the next byte. This heuristic, when augmented to LS and RT-based techniques, may lower down the false negatives by finding all possible instructions in the “gap” area.

3 PE Architecture and Intel Instruction Format

The path from a binary to disassembly consists of several steps that convert sequence of bytes to assembly instructions such that it can be easily analyzed. Given a sequence of bytes the conversion of bytes to intermediate representation depends upon the choice of processor architecture. All architectures have certain opcode map for mapping bytes sequence to instructions and vice versa. For our purposes we use Intel architecture [3]. In order to reverse engineer a binary executable we need to understand the steps involved in it. Since most of the virus attacks target PE (win32 portable executable) files, we analyze and disassemble only PE files (although the methods proposed should be readily extendible to DOS executables, ELF).

This chapter introduces:

- PE File format,
- Intel instruction format [3, 18], and the
- Intel Opcode Map [3]

3.1 PE file format

Fig. 3-1 shows the general layout of a PE file. All PE files starts with DOS MZ header followed by a small program called DOS stub. The stub is the small program that gives out an error message if the program cannot be run in DOS mode. Here we only discuss the fields that are required to get the information of the byte sequence to be disassembled. One of the important parts of PE file is the header that contains the time and date stamp of PE file, the size of code, the address of entry point etc. Other important parts to focus in PE files are the section headers contained in the section table. PE executables are divided into chunks of bytes called “sections.” Each section has a

different attribute such as whether it is code or data or whether it is read or write enabled.

The number of entries in the section table is dependent upon the total number of sections

i.e. if there are n number of sections there will be n entries in the section table.


<i>DOS “MZ” header</i>	Contains the magic number “MZ” (0x 4d 0x 5a) in first two bytes. Last byte contains offset of PE header
<i>DOS STUB</i>	Contains small program saying “this program cannot be run in dos mode”.
<i>“PE” header</i>	Contains a magic number PE00 as starting 4 bytes. Info such as number of sections, entry point, image base, time and date stamp etc
<i>Section Table</i>	Contains Headers for each section. Each Header contains information for sections like size, RVA, name and characteristics stating it to be code or data section.
<i>Section 1</i>	 <p>Sections of PE file containing byte code. More than one section can have same characteristics.</p>
<i>Section 2</i>	
<i>...</i>	
<i>Section n</i>	

Fig. 3-1. File format of a PE file.

It is not necessary that a section be labeled as “data” or “code” since section can contain both data as well as code. For disassembling, we need the sections and the information about sections, such as virtual address of the section, size of the section, and its real address in the file. All information about sections can be found in section header.

3.2 Intel Instruction format

Some knowledge of the instruction format of Intel is required to convert byte sequences into meaningful assembly instructions. Intel instruction can vary in length from one byte to fourteen bytes. The structure of every instruction is fixed, and illustrated in the Fig. 3-2 [18].

Instruction	Prefix	Opcode	Mod R/M	SIB	Displacement	Immediate
No of bytes	0 – 4 bytes	1 or 2 bytes	0 or 1 byte	0 or 1 byte	0 – 4 bytes	0 – 4 bytes

Fig. 3-2. Intel instruction format

An instruction can start with an optional prefix of up to four bytes. Instruction prefixes modify default instruction behavior, i.e., they can force default segment of an instruction or can override default size of machine word, to either 16 or 32-bit word. Prefix also determines whether the following opcode is of one byte or two bytes. Specifically, if prefix is “0x0F” the opcode is a two-byte opcode. An opcode decides the primary format of an instruction. It determines whether there is further byte following the opcode or the instruction is a one-byte instruction. An opcode also determine about direction of operation.

If required, optional bytes, such as Mod R/M, SIB, displacement, and immediate can follow the opcode byte. Mod R/M byte decides the addressing mode (direct/indirect etc) of the operand. It specifies about the register to be used in the instruction and indexing modes. Optional byte SIB (Scale Index Base) might follow Mod R/M byte to extend the different addressing modes represented by Mod R/M. Fig. 3-3 shows the architecture of both Mod R/M byte and SIB byte [4, 18]. Following formula is used to compute the SIB value: “(index *2^{scale}) + base” and used for representing complex addressing modes.

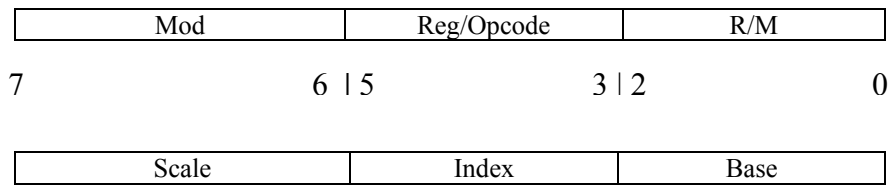


Fig. 3-3. Architecture of Mod R/M byte and SIB byte

Example showing interpretation of an instruction:

Mod byte is a two-bit value (most significant bits in Fig. 3-3). It can have 4 possible values 00, 01, 10 or 11. Which of the optional byte follows Mod R/M is decided by the Mod R/M value. For example, the combination 01 or 10 in the Mod field leads to displacement byte following the Mod R/M byte. Lets consider an example.

Prefix	Opcode	Mod R/M	SIB	Displacement
----	10001011	00000100	10000101	01001100 01111011 01000010 00000000
Instruction: <i>mov eax , dword[4*eax+427b4c] = 8b04854c7b4200</i>				

Fig. 3-4. A 32 bit instruction and its structure

Fig. 3-4 illustrates disassembly of a sequence of bytes to an assembly instruction. The opcode value specifies that the instruction is a *mov* instruction. The X86 processor and memory manager uses mapping tables to map opcodes, Mod R/M bytes and SIB bytes to corresponding tables and interpret any instruction. This is an interesting example because the Mod field is neither 01 nor 10, yet there is a displacement byte following it. This example highlights the need for opcode maps, containing detailed rules, to map byte-sequences to instructions.

3.3 Intel Opcode Maps

X86 processors divide the instructions into three encoding groups, namely: 1-byte opcode encodings, 2-byte opcode encodings, and escape (floating-point) encodings [3]. Section 3.2 discussed how prefix value affects length of opcode. If the opcode is not preceded by prefix “0x0f” then it is a single byte opcode, and if the opcode is preceded by “0x0f” prefix then it is a two-byte opcode. Each opcode has a lookup map associated with it. For example, if the opcode is 0x08, then the associated table entry will be one of ADD, OR, ADC, SBB, AND, SUB, XOR or CMP. If it is “0x0f 0x80” then the two-byte opcode map is referenced instead and the instruction is *jo* (jump on overflow) in this case.

The opcode map contains all the information about the byte following the opcode if it is part of instruction. Specifically, for each value of opcode the map contains information about register type, whether Mod R/M byte follows or not etc.

3.4 Addressing modes

X86 normally supports two addressing modes for its operands: direct and indirect. It also provides unconditional and conditional branch or jump instructions. Examples of conditional jump instructions are *jz*, *je*, *loopne* etc and examples of unconditional jump instructions are *jmp*, *call*, *ret* etc. Both conditional and unconditional branches can jump either directly or indirectly. Example of direct *jmp* is *call 401000* where we know the exact address that it jumps to, in case of indirect jump it can jump via registers as in *call eax*.

4 Segmentation Overview

A key contribution of this thesis is an algorithm for extracting maximal set of valid instructions embedded in basic blocks (termed as segments). The key idea behind segmentation is to find all the possible code in the executable by considering each byte as potentially starting an instruction. The basic steps of segmentation algorithm are:

- Exhaustively check each byte for starting an instruction
- Create segments – sequences of instructions
- Chain segments – link segments based on jump targets
- Prune segments – remove segments leading to invalid instructions

The segmentation approach does not rely on certain common assumptions like entry point information and control flow information.

This chapter elaborates on the following points:

- Segmentation efficacy
- Segment structure
- Segment relationships
- Segment construction
- Overlap special cases

4.1 Segmentation Efficacy

In most prior disassembly approaches, it was effectively assumed that no byte may belong to more than one instruction, and that a binary is effectively a contiguous collection of non-overlapping instruction sequences. A key question to ask, however, is:

Can one byte be in two instructions?

Consider the byte sequence and control flows depicted in Fig. 4-1 and Fig. 4-2.

In the figures arrows indicate the start and ending bytes of instructions.

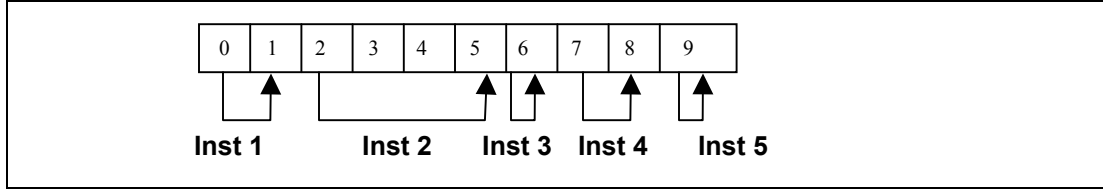


Fig. 4-1. Example showing control flow of program starting at byte 0

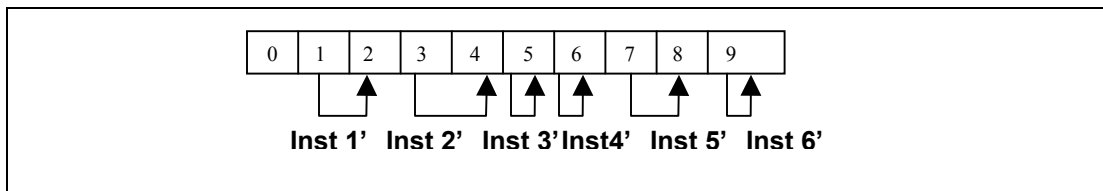


Fig. 4-2. Example showing control flow of program starting at byte 1

Assume Fig. 4-1 represents what is considered to be a “correct” instruction set of the program. If the disassembly is off by one byte due to incorrect disassembly of data [17], we might get a different instruction set as shown in Fig. 4-2. Since, 398 out of possible 512 opcodes in the Intel instruction set are valid, therefore *Inst 1'*, *Inst 2'*, and *Inst 3'* could potentially start valid instructions. Fig. 4-2 shows an instruction set starting from byte 1. It can be seen that the disassembly “self-repairs” [17] itself at byte 6. The existing disassembly algorithms would consider Fig. 4-2 to be the correct disassembly, thereby missing the control flow of Fig. 4-1.

To overcome this limitation, it is necessary to consider each byte as potentially starting an instruction. In our approach each byte could potentially start an instruction, thereby presenting an exhaustive way of finding all possible instructions. Hence, it is possible to find all disassembly paths in the binary. The pool of instructions generated by

analyzing all the bytes gives the largest possible instruction set within the binary, i.e., the algorithm cannot have any false negatives.

4.2 Segment Structure

Fig. 4-3 depicts the structure of a segment. Each segment is the maximal sequence of instructions and is terminated by an instruction that represents transfer of control such as *jmp*, *call*, *loopne* etc.

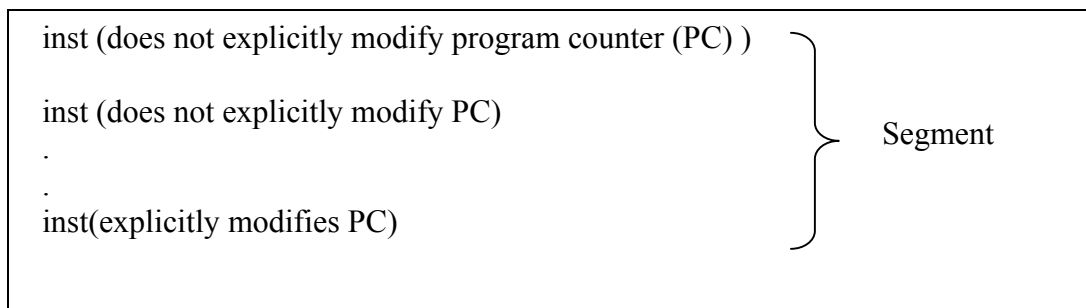


Fig. 4-3. Structure of a segment

All instructions implicitly increment the program counter by one address. We say that an instruction explicitly modifies program counter (PC) if it is a control transfer operation, i.e., PC does not point to the byte immediately after the instruction. For example *jmp*, *call* and *ret* explicitly modify PC.

4.3 Segment Relationships

Since each byte may theoretically start an instruction, it may also start a segment. The byte starting a segment is denoted as *segStart*; the byte ending a segment is denoted as *segEnd*. A unique segment id *segId* is assigned to each segment.

The *segStart* and *segEnd* are actual locations of bytes in a binary file. For any two segments S_i and S_j , $S_i = S_j$ iff $i = j$, for all $i, j \geq 0$. If $j > i$, then the starting byte of S_i is less than starting byte of S_j , i.e., in the binary file, segment S_i starts prior to segment S_j . We

call each segment a “potential segment”, until it is *chained successfully* (section 5) to other potential segments, to regenerate the assembly code. These potential segments can be categorized according to their properties as shown below.

1. *Overlapping segment*: For any S_i, S_j , such that $j > i$, $S_j.segStart < S_i.segEnd$ implies S_i and S_j are *overlapping segments*.
 $\forall i, j$, such that $j > i$ and $S_j.segStart < S_i.segEnd \Leftrightarrow \text{overlapping}(S_i, S_j)$
2. *Intermediate segment*: S_j is an intermediate segment of S_i iff $j > i$, and S_i and S_j are overlapping segments. Exactly one instruction is common in both segments. and $S_j.segEnd < S_i.segEnd$.
 $\forall i, j, j > i$ and $\text{overlapping}(S_i, S_j)$ and $|S_i \cap S_j| = 1$ and $S_j.segEnd < S_i.segEnd \Leftrightarrow \text{Intermediate}(S_j, S_i)$.
3. *Master Segment*: Segment S_i is called a master segment if there exists a segment S_j , such that S_j is an intermediate segment of S_i .
 $\exists S_j. \text{Intermediate}(S_j, S_i) \Rightarrow \text{masterSegment}(S_i)$

4.4 Segment Construction

Fig. 4-4 depicts the steps for segment creation starting from the byte B0. In the figure the bytes shown in gray color are not analyzed for possible instruction start. S0 and S1 are intermediate stages of segment creation, while S2 is the final segment.

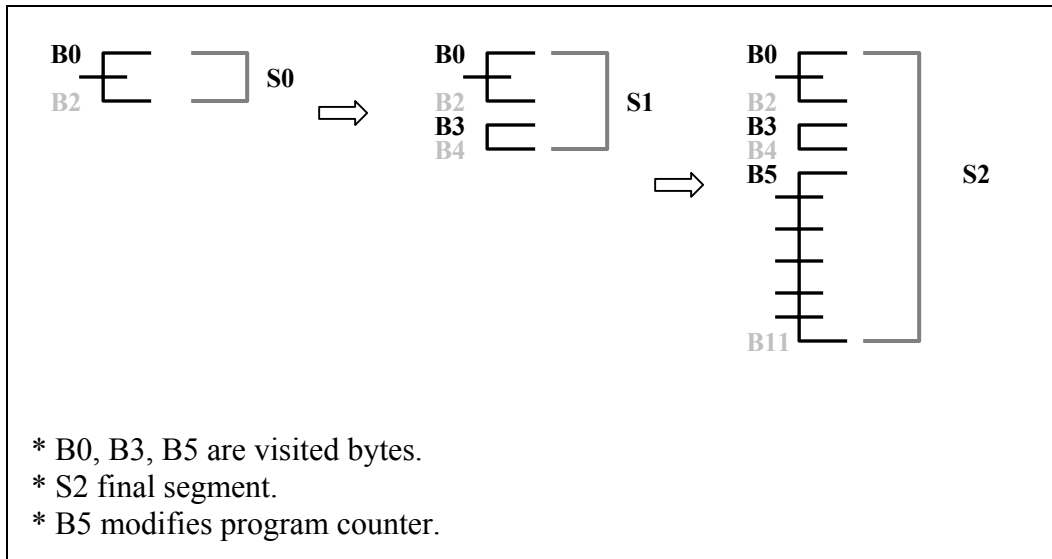


Fig. 4-4. Steps of segment creation

Each byte that starts an instruction may lead to a segment that is either valid or invalid. Fig. 4-5 shows the life cycle of segment creation from *S1* to *S4*. *B0*, *B1* and *B2* start valid instructions (shown by black brackets). The segment is represented by gray bracket, in its various stages of development. Depending upon byte *B3*, the segment is termed either a valid potential segment or an invalid/corrupt segment. If *B3* is the byte that starts a valid instruction then *S4* is a valid potential segment. In case *B3* does not start any instruction *S4* is an invalid/corrupt segment.

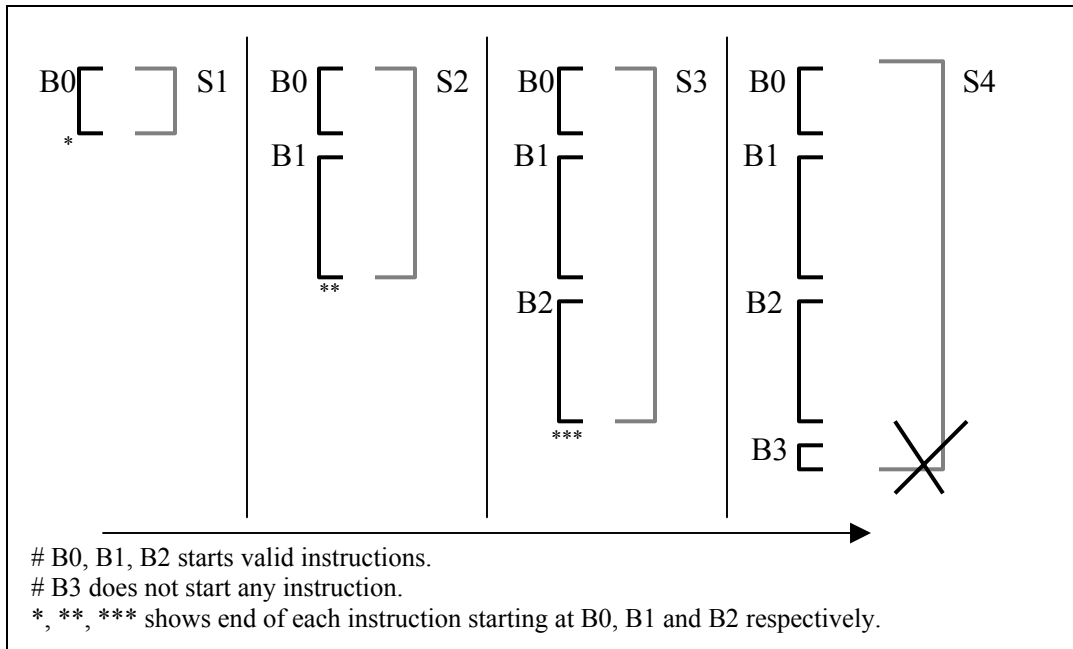


Fig. 4-5. Segment termination

The algorithm is shown in Fig. 4-6, and uses the following keywords and methods:

potentialSegmentSize(b) : The algorithm assigns a value corresponding to each byte b called *potentialSegmentSize(b)* or *PSS(b)* (read as potential segment size of byte b). PSS of each byte is overloaded and has different meaning depending upon its value. There are three cases for this

- Case 1.** *potentialSegmentSize(b)=0* implies this byte does not start a valid segment, i.e., a segment that has only valid instructions.
- Case 2.** *potentialSegmentSize(b)=n*, where $n > 0$ implies, bytes b to $b + n - 1$ form a potential segment.
- Case 3.** *potentialSegmentSize(b)=n*, where $n < 0$ implies the bytes from $b + n$ (which is less than b , since $n < 0$) to $b + n + \text{potentialSegmentSize}(b + n) - 1$ is

a potential segment, if the *potentialSegmentSize(b+n)* is not equal to zero.

getPotentialSegmentSize (byte → Int) gives the number of bytes following upto which, there are, sequence of valid instructions that do not explicitly modify the program counter (PC).

setPotentialSegmentSize (byte, int → void) sets PSS value corresponding to each byte.

We would typically need the longest “potential segment”.

masterSegmentStart (segment, int → void) represents the start of a master segment of an intermediate *segment*.

```

computePotentialSegment (b, size)
{
  if (!isVisited(b)) {
    setVisited(b, true);
  }

1.  if (getPotentialSegmentSize(b) < 0)
    {
2.    if(getPotentialSegmentSize(b+getPotentialSegmentSize(b))==0)
      {
        /* corrupt segment. Delete ongoing segment and exit */
        setPotentialSegmentSize ((b-size), 0);
3.    } else { /* instruction part of segment terminate ongoing segment and return */
        setPotentialSegmentSize((b-size), size);
        masterSegmentStart(this,b+getPotentialSegmentSize(b));
      }
    } else if (!isStartsInstruction(b) ) {
      /* corrupt segment. Delete ongoing segment and exit */
4.    setPotentialSegmentSize((b-size), 0);
    } else if (modifiesPC(b)) {
      /* end of segment found */
5.    setPotentialSegmentSize(b, -size);
      /* mark beginning of segment at b-size */
      setPotentialSegmentSize((b-size),(size + instructionSize(b)));
    } else {
      setPotentialSegmentSize(b, -size);
6.    local_size=instructionSize(b);
        computePotentialSegment(b+local_size, size+local_size);
      }
}

```

Fig. 4-6. Pseudocode for segmentation algorithm, Leftmost column specifies code waypoints

<pre> main() { Segment segArr[]; for (i = 0; i < length(program); i++) { setVisited(i, false); setPotentialSegmentSize((i), 0); } segArr=new Segment(length(program)); for (i = 0; i < length(program); i++) { if (!isVisited(i)) { segArr[i].computePotentialSegment(i,0); /* if segment exists */ segArr[i].segStart= i; segArr[i].segId=i; } } } </pre>	<pre> Class segment { int segStart=-1; int segEnd=-1; int segId=-1; int masterSegment=-1; } </pre>
---	--

Fig. 4-7. Main program pseudocode for computing segments

4.5 Explanation of algorithm waypoints

The segmentation algorithm is a recursive process. The objective of the segmentation algorithm is to divide the disassembly into chunks of valid segments, where each segment represents a sequence of valid instructions. The algorithm starts the disassembly from a particular starting byte of the program (in the case of PE file it is the first byte of code section) and tries to form segments. In the process of creating a segment, each byte encountered will be flagged as visited. The algorithm analyzes each byte for the following properties.

- Checks whether the byte start an instruction or not.
- If the byte “*b*” starts an instruction “*I*”, then the algorithm checks for one of the three properties for the instruction “*I*” to hold.

1. If the instruction “*I*” modifies the PC explicitly, the byte is marked as an end of segment.
2. If “*I*” does not modify the PC, it is examined to determine if it is part of any existing segment. If “*I*” belongs to any existing segment, the byte “*b*” is marked as last byte of the segment. The current segment is marked *intermediate segment* and the existing segment is marked as the *master segment*.
3. If the instruction is neither part of any existing segment nor it is a PC-modifying instruction, it is added to the current segment under formation. For example, instructions starting from B0, B1, and B2 in Fig. 4-5 are all added to segment S4, which is under formation.

The program visits each byte only once; this makes the analysis of program linear, depending upon file size. The program disassembles linearly until it hits a data byte, or an instruction that modifies PC or any instruction that is part of other existing segments. The left-most column of Fig. 4-6 depicts the algorithm waypoints.

- **Waypoints 1, 2, 3.** A negative potential segment size value of a byte represents that it had been visited. If it is visited, there are two cases: either the byte is part of any existing segment or it is a data byte. A byte is a data byte if it does not start an instruction or is a part of a corrupt segment. If the current byte being examined is part of an already existing segment, then the byte as well as the segment under formation, is designated a master segment. The current byte is then marked as the end byte of the segment.

- **Waypoint 4.** If the byte being analyzed in the current segment does not start an instruction, the potential segment size associated with the byte starting the segment (say B0) is assigned zero. This denotes that B0 does not start any segment.
- **Waypoint 5.** If we encounter a byte B1 that starts an instruction that explicitly modifies PC, we assign the byte B0 the length of segment equal to the address offset between B0 and B1. Positive value at B0 denotes that B0 start a segment of specified positive length.
- **Waypoint 6.** The program is called recursively until the end of segment is found or it hits a data byte. The negative PSS value assigned to a byte denotes that byte does not start a segment rather is a part of some segment.

Here, each byte is analyzed only once therefore the complexity of the algorithm is $O(N)$, where N is the number of bytes in the program.

4.6 Segment Overlaps and Special Cases

While analyzing every byte of the program for starting a segment, the analysis may generate many overlapping segments that may or may not be part of the code. To understand an overlapping segment, let's consider the example in Fig. 4-8. In the figure, $B.\text{segStart} < A.\text{segEnd}$, where A and B are overlapping segments (see section 4.2). A starts at byte 0 while B starts at byte 1 (shown with curly braces). The actual control flow of the program is shown with black arrows, while the gray arrows represent secondary control flow, starting from byte 0. Byte 0 is a data byte that generates an instruction along with byte 1.

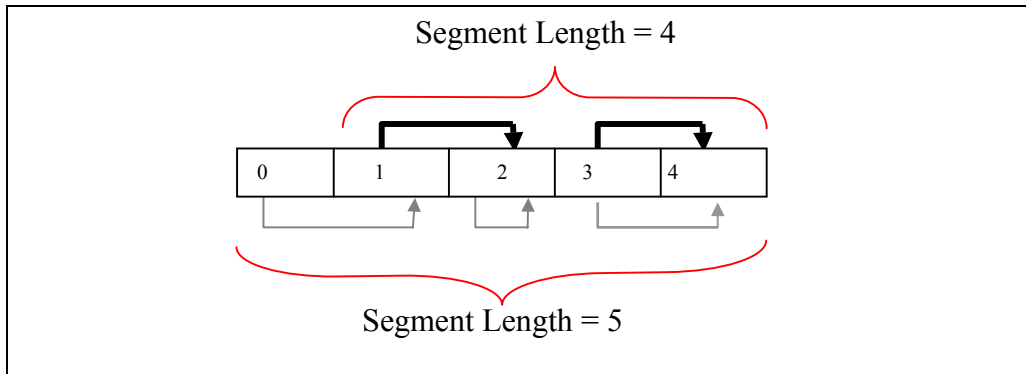


Fig. 4-8. Illustrative example of segment overlaps

There are several cases that may occur when segments overlap:

1. Fig. 4-9 illustrates the case where B is merging into A at point I' , where I is not an instruction of A . If the first instruction I in Segment B is also an instruction of Segment A , then segments A and B must end in same byte. In this case the algorithm would consider A as primary segment and discard B .
2. If I is not an instruction of A then B and A are Overlapping Segments. The first instruction where B “self-repairs” [7] and merges into A , is the end of segment B (I' in Fig. 4-9). B in this case is an intermediate segment while A is a *master segment*.
3. In case B merges into an invalid segment A , B is also marked as invalid.

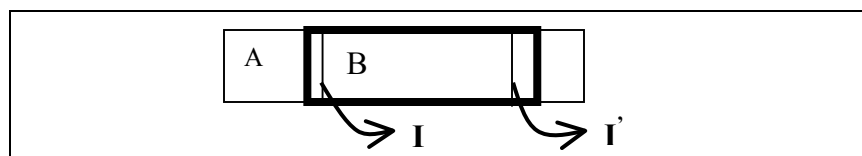


Fig. 4-9. Example showing special case of overlapping

4.7 Illustrative Example

Fig. 4-10 shows an example byte sequence and the target assembly. Fig. 4-11 shows the disassembly generated by our algorithm.

Memory Byte	Hex Equivalent of Instruction	Assembly Code
401000	b8 04 00 00 00	mov eax,4
401005	83 c0 06	add eax,6
401008	a3 33 20 40 00	mov x,eax
40100d	01 05 37 20 40 00	add y,eax
401013	c7 05 3b 20 40 00 00 00 00	mov counter,0
40101d	b8 30 10 40 00	lea eax,label1
401022	ff e0	jmp eax

Fig. 4-10. Example 2: excerpt of assembly code from a sample program

The above code is an initialization code that finally jumps to *label1* to start the processing. This code excerpt is a simple indirect jump. To disassemble the code, *label1* might be calculated using data flow analysis.

Fig. 4-11 shows the segments that are created after using the segmentation algorithm. All the segments are numbered in ascending order according to their starting address. Segments 1 and 8 are normal segments that end with instructions that modify the PC explicitly. Grey blocks denote end of a segment.

Whenever any two segments have the same ending instruction, they form an intermediate-master segment pair. For example, segment 2 is an intermediate segment of segment 1, it meets segment 1 at location 401005. Segment 2 is thus an intermediate segment of segment 1. Similarly segments 3, 4, 5, 6, 7, and 9 are intermediate segments to segment 1. Although, segment 8 overlaps with segments 1, 7, and 9 it is not an intermediate segment of any of them. If the disassembly had been off by 2 bytes (due to any disassembly error), the disassembly would start at 401002 and continue until

instruction 40100d (self-repairing point). In such a case, the true instructions are missed if any of the linear sweep or recursive traversal methods is used; the segment algorithm gives all the possible sequences of code. It thus leaves no code behind from disassembling. Also, note that each byte can belong to up to 4 segments in this case.

Address (Bytes)	Hex	Segment 1	Segment 2	Segment 3	Segment 4
401000:	b8	<i>mov eax, 4</i>			
401001:	04				
	00				
	00				
	83				
	c0				
	06				
	a3				
	33				
	20				
	40				
	00				
	01	<i>add dword[402037], eax</i>			
	05				
	37				
	20				
	40				
	00				
	07				
	05				
	3b				
	20				
	40				
	00				
	00				
	00				
	00				
	b8				
	30				
	10				
	40				
401021:	00				
401022:	ff	<i>jmp eax</i>	<i>jmp eax</i>	<i>inc eax</i> <i>add bh, bh</i>	<i>jmp eax</i>
401023:	e0			<i>loopne near ptr 401080</i>	

Fig. 4-11. Effect of applying segmentation algorithm on given code block, Grey area shows end of segment.

5 Segment Pruning via Segment Chaining

The segmentation algorithm divides the disassembly into chunks of segments containing *valid* instructions. It finds all the potential valid segments in the code and discards the invalid segments. However, overlapping segments can generate false positives. Segmentation deletes some segments that lead to invalid instructions; however, in order to delete the potential segments that lead to invalid segments, other methods are required. We use *chaining* to delete the segments that cannot logically be real. This allows regeneration of precise assembly. This chapter elaborates on the following points:

- Segment Chaining
- Segment Pruning

5.1 Segment-Chaining

A segment can either terminate at an instruction that explicitly modifies the PC, or it can be an intermediate segment. In the former case, it is possible to chain the segments through direct *jumps* or *calls*. For example when a segment A ends with an explicit jump (direct jump) to segment B. It can be chained to the beginning of B. We then say, “*Segment A points to segment B*” or “*Segment A leads to segment B*” or “*segment A points to byte b*” (byte b does not start a segment).

While chaining a segment *S*, if some segment in the path is pointing to a bad byte or data byte (i.e. byte does not start a segment) we discard *S*. We also discard all the segments pointing to *S*. Fig. 5-1 depicts various stages of segmentation. The chaining stage shows segment chaining by direct jumps. While the pruning stage shows the result

of pruning in case the jump target of some segment is a corrupt byte or byte that does not start any segment.

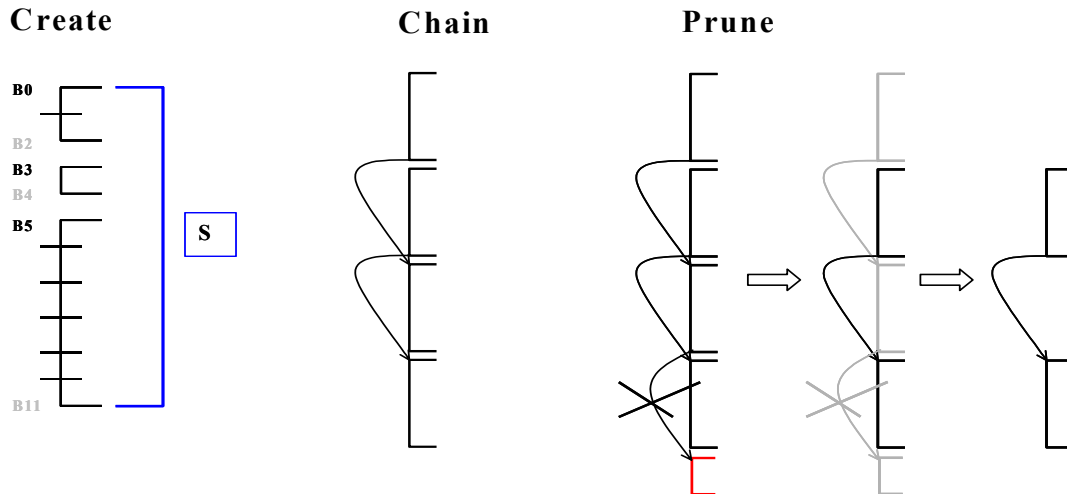


Fig. 5-1. Various stages of segmentation

Since we make an intermediate segment point to its master, the intermediate segment points to the same target locations where its master is pointing. If the master segment is pruned in the chaining process, we delete all the intermediate segments of this master segment. In a close to linear traversal of the segments, we can identify and discard most potential segments that lead to the non-instruction byte. Fig. 5-2 and Fig. 5-3 show the algorithm to connect all the segments as “one chunk” and it prunes all the segments that lead to a non-instruction.

5.2 Algorithm: Chaining and Pruning

Chaining finds all the segments and connects them through edges determined by direct jump targets. In case the target location of a segment S is corrupt or an invalid byte S gets pruned, then all the segments pointing to S also gets pruned. The segment can end by any one of the four types of instruction:

- Conditional direct jump,

- Unconditional direct jump,
- Indirect jump (not handled by chaining algorithm) or
- Any arbitrary instruction (Segment is an intermediate segment)

The following steps trace through the waypoints shown in the Fig. 5-2.

1. The positive PSS value v for any byte b denotes start of a segment of length v .
2. If the last instruction of a segment S is either the conditional or unconditional direct jump, the target location is verified for starting a valid segment S' . In case S' is a valid segment S is chained to starting location of S' . Otherwise S is discarded.
3. In case the last instruction of a segment S is a conditional jump, the next byte ($jbNext$) to the branching instruction is also verified along with jump target (jb) for potentially starting a segment S'' . If it starts a segment then S and S'' are chained. If both jb and $jbNext$ do not start valid segments, then S is discarded.
4. If the last instruction of a segment S is any arbitrary instruction that does not explicitly modify PC, it implies that S is an intermediate segment of a master segment M . In such a case, we chain S with starting location of M .

```

for (i = 0; i < length(program); i++)
{
  jbNext = -1;
  1. if (getPSS(b) > 0)
  {
    if (last instruction of Segment(b) is a direct jump )
    {
      jb = address of the target of the jump;
      if (conditional jump)
      { jbNext = next byte after segment(b); }
      2. if (validAddress(jb))
      {
        if (getPSS(jb) == 0 && jbNext == -1)
        deleteSegment(b);
        else if ( getPSS(jb) > 0 )
        create an edge from b to jb
        else if (getPSS(jb) < 0 )
        create an edge form b to jb + getPSS(jb)
      }
      3. if (validAddress(jbNext))
      {
        if (getPSS(jb) == 0 && getPSS(jbNext) == 0)
        deleteSegment(b);
        else if ( getPSS(jbNext) > 0 )
        create an edge from b to jbNext
        else if (getPSS(jbNext) < 0 )
        create an edge form b to jbNext + getPSS(jbNext)
      }
      if (! (validAddress(jb) && validAddress (jbNext)))
      deleteSegment(b);
    }
    4. else if ( last instruction of Segment(b) does not modify PC ) {
      create an edge from b to masterSegment(jb);
    }
  }
}
}

```

Fig. 5-2. Pseudocode for segment-chaining algorithm, Leftmost column specifies code waypoints.

```

deleteSegment(b)
{
  for each segment i having edge to segment b
  { deleteSegment(i); }
  Segment(b)=null;
}

```

Fig. 5-3. Pseudocode for deleting a segment

5.3 Example: Segment Chaining and Pruning

Fig. 5-4 shows some cases of segments and how the chaining and pruning is done to these segments to reduce the false positives. The figure is showing chaining and ripple effect in deleting the invalid segments when a segment hits a bad byte.

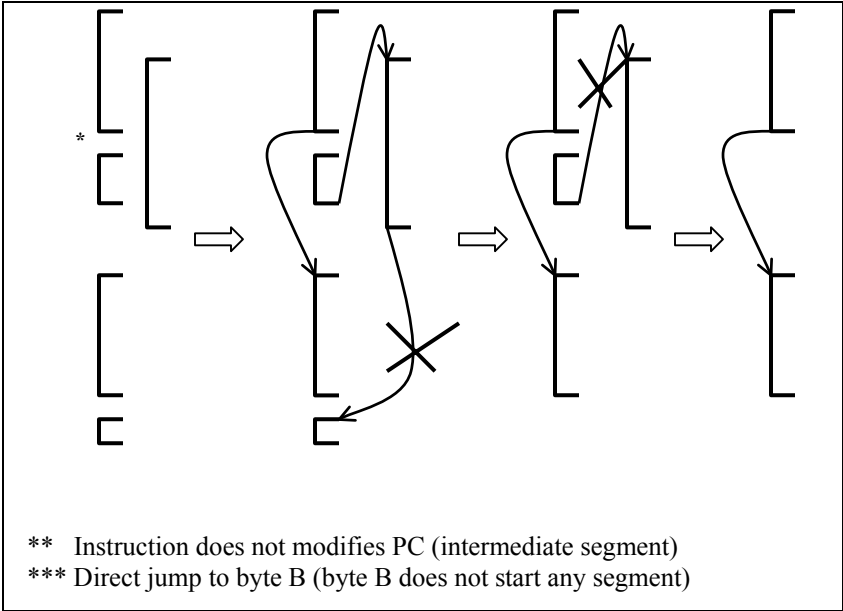


Fig. 5-4. Example showing segment pruning by chaining

Fig. 5-4 depicts a segment in its various stages of chaining. A deletion of an edge leads to deletion of the segment. Deletion is marked by symbol X on the edges.

6 Evaluation and Results

This section presents the performance evaluation of the proposed algorithms and highlights the limitations faced by them. The experimental setup for the evaluation is also presented. The main aim of this experiment is to study the percentage of valid instructions and the percentage of reducible false positives in a given binary, after applying the proposed algorithm. We divide the binaries into three different classes: first is class of clean Windows™ executables (e.g. notepad.exe), second is a class of malicious executables, and finally a class of non-executable[†] files. The evaluation measures of interest are: number of valid instructions in a binary (baseline), amount of false positives with respect to baseline, effect of chaining/pruning on these measures, and amount of code inflation.

6.1 Theoretical Limitations and Assumptions

In the game of obfuscation/deobfuscation, the fewer assumptions the disassembler makes, the harder it is for hostile programmers to attack the static analysis. The main threat in the context of malice is missing malicious code. False positives might give an imprecise disassembly, but false negatives are the real threats, that give unsafe disassembly. Our proposed solution does not give any false negatives, if the binary satisfies certain assumptions.

6.1.1 Attacking the Assumptions

This subsection discusses the limitations of our solution, and how malicious code writers can attack it. Let's look at the code snippet from worm Netsky.Z

[†] Non-executable files include image files as well as compressed files.

- Self-modifying code:** The example in Fig. 6-1 shows code that self modifies itself as it executes. Self-modifying code can cause disassemblers to miss some code because the instructions are generated at runtime, so what once was data could become code in the future and vice versa. Prior algorithms silently fail after giving few disassembled instructions as evident from Fig. 6-1, which shows disassembly results from an open source debugger Ollydbg. It stops disassembling after location 00403E6F. Disassembling starting at each byte may miss only the instruction starting at 00403E6E. However, self-modifying code can increase false negative manifolds if the code shown is repeated or tweaked slightly and then repeated. The segmentation algorithm cannot handle this attack.

Location	Column 1		Column 2	
	Hex	Disassembly	Hex	Disassembly
00403E5F	B8 6E3E4000	MOV EAX, 00403E6E	B8 6E3E4000	MOV EAX, 00403E6E
...				
00403E64	8000 28	ADD BYTE PTR DS:[EAX],28	8000 28	ADD BYTE PTR DS:[EAX],28
...				
00403E67	40	INC EAX	40	INC EAX
00403E68	8100 67452301	ADD DWORD PTR DS:[EAX],1234567	8100 67452301	ADD DWORD PTR DS:[EAX],1234567
...				
00403E6E	90	NOP	B8 32BC5C00	MOV EAX, 005CBC32
00403E6F	CB	RETF		
00403E70	76	DB 76		
00403E71	39	DB 39		
00403E72	FF	DB FF		
00403E73	50	DB 50	50	PUSH EAX

Fig. 6-1. Runtime self-modifying obfuscation (Netsky.Z)

- Structured exception handling (SEH).** Exception handling in windows is like a try catch block in Java where whenever an exception is raised in try block it is caught in the catch block. In Windows™ programming the programmer can associate a handler to each exception. Whenever any particular exception is raised, the corresponding handler code is invoked. Exception handling can also be

used to confuse the jump target. Malicious code writer can craft an exception to confuse disassembler inserting the malicious code at the location of the handler. Neither the current algorithms nor the proposed algorithm can deal with such a situation. Fig. 6-2 gives an example of exception handling used to hide malicious code.

Location	Hex	Disassembly
00403E5F	B8 6E3E4000	MOV EAX, 00403E6E
...		
00403E64	8000 28	ADD BYTE PTR DS:[EAX],28
...		
00403E67	40	INC EAX
00403E68	8100 67452301	ADD DWORD PTR DS:[EAX],1234567
...		
00403E6E	B8 32BC5C00	MOV EAX, 005CBC32
...		
00403E73	50	PUSH EAX
00403E74	64FF35 000000	PUSH DWORD PTR FS:[0]
00403E7B	648925 000000	MOV DWORD PTR FS:[0],ESP
00403E82	33C0	XOR EAX,EAX
00403E84	8908	MOV DWORD PTR DS:[EAX],ECX
00403E86	90	NOB

Fig. 6-2. Obfuscation through exception handling

At location 00403E82 the virus writer set *eax* to 0 and the next instruction at location 00403E84 accesses the memory location 0, this will raise a divide by zero exception that is handled by the handler at location 005CBC32. At this handler location the virus writer puts the malicious code hidden from all disassemblers.

6.1.2 Attacking the Implementation

Code outside the code sections: In ordinary binaries, code is normally restricted to the code sections. However, in the case of viruses and worms code may generally be contained in data sections. Our current implementation assumes that all the code is found in the code sections and code in data sections will not be found.

Alignment byte spoofing: Alignment bytes are frequently used to align code to page boundaries. They are commonly found either in the form of 0x00s or sequence of 0xCCs. However, 0xCC is also a one byte instruction *int 03*, and 0x00 followed by another 0x00 makes a two byte instruction *add byte[*eax*]*. Our current implementation assumes that after X bytes in succession, the bytes form either a data block or an alignment block. Hostile programmers could craft extra byte sequences using these alignment bytes, so that the disassembler assumes it to be a data block and does not disassemble it. Nonetheless, the threshold length is an implementation-specific parameter that can be changed, so it could easily be increased to catch the instruction, although doing so may result in increased false positives on data blocks. On a positive note, attacking disassemblers in such a way might greatly increase the size of the executable, something, which hostile programmers generally try to avoid.

6.2 Experimental Setup and Analysis

The aim of this study is to analyze disassembly results when the proposed algorithm is applied on different classes of binaries or data files. The results would be used to understand the differences between clean, malicious and non-executable data files.

6.2.1 Setup

For analysis of a binary file, we need a disassembler and a collection of binaries. We developed a Java based plugin-disassembler using Eclipse PDE (Plugin development environment). To parse the PE header, we attached C code provided by Sang Cho [10] to the disassembler using JNI [8, 12, 16]. In order to parse the instructions, Intel instruction information is gathered from Intel Architecture Software Developer's Manual [3].

6.2.2 Analysis Parameters

The results presented here are the statistical outcome of multiple runs of segmentation algorithm, on different pools of binary files. The following parameters are calculated to study the results.

- I. *Total Valid Instructions (TVI)*: The instructions disassembled before the segmentation algorithm is applied are *Total Valid Instructions* in a binary. TVI indicates maximum number of instructions possible in a binary starting from each byte, excluding alignment bytes. This parameter shows the statistical differences of TVI depending upon file size and file class.
- II. *Reduction in false positives (RFP) before and after chaining*: Segmentation algorithm divides the program in segments. The instructions that are not part of these segments are invalid. This leads to reduction of instructions from initial set TVI. Let $R1$ represent total number of instructions in valid segments, where, $R1$ is a subset of TVI. The ratio $1 - (R1: TVI)$, depicted by $R2$, gives the percentage reduction in the instructions due to segmentation.

Chaining further removes the valid segments and hence further reduction of valid instructions from the set $R1$. The subset of $R1$ thus created is termed $R1'$. $R1'$ and $R2'$ represent the results due to chaining. The instruction count information before and after segmentation shows the reduction in target locations of possible indirect jumps and gives the estimate of code that has to be analyzed even if brute force approach as this one is used. It is very hard to hide instructions by using junk insertion or jump in the middle of other instructions as we analyze each byte using brute force.

III. *Segment Pruning (SP) through chaining*: “Number of segments present in the code,” is another metric for analysis of performance of proposed algorithm. The more segments are removed during chaining, the more unique disassembly is generated.

$R3$ represents the total number of valid segments that are created after segmentation, whereas $R4$ is the percent of valid segments in the binary. Only instructions in the set TVI take part in creating segments. The percentage of valid segments is the ratio between valid and total segments wherein total segments contains both valid and invalid segments. *Note that Invalid segments are segments terminated due to hitting some data byte during initial segmentation.* $R3'$ represents the decrease in segments due to segment chaining. If the decrease is not significant, we would require more heuristics measures to further remove the invalid segments.

IV. *Code Inflation Index (CII)*: Since dealing with each byte for the possibility of instruction start, it is possible that a byte may belong to more than one instruction and, thus, more than one segment. Whenever a byte belongs to more than one segment, it is counted twice towards the code. Such inflation of code might provide interesting statistical data. The ideal target of algorithm is to keep the results close to one so that no one byte can finally belong to more than one instruction. In that case, we get a unique disassembly. CII or Code Inflation Index represents the number of segments to which each byte belongs. Only the bytes present in segments duplicate the code. So, if the file size is 10 bytes and if CII is two, it does not mean that the disassembled code size is 20 bytes. However, if

seven out of ten bytes are present in valid segments, then the disassembled code size will be 14 bytes with value of CII equal to two. R5 and R5' represents the values of CII before and after chaining.

6.2.3 Data Analysis: “Clean” Executable

The following collection of windows executables were tested for the above discussed analysis parameters: cat.exe, cut.exe, gcc.exe, grep.exe, javac.exe, notepad.exe, eclipse.exe, calc.exe, winmine.exe. The result for these binaries is shown in Table 6.1.

Table 6-1. Results: Clean Executable (instruction, segments, and overlaps)

Filename (.exe)	File Size (In bytes)	Total Valid Instructions (TVI)	Instructions (RFP)				Segments (SP)				Code Inflation (CII)	
			Before chaining		After chaining		Before chaining		After chaining		Before chaining	After chaining
			R1	R2	R1'	R2'	R3	R4	R3'	R4'	R5	R5'
Cat	17,408	13,070	10,761	82.33	8,747	66.92	3,697	81.11	3,090	67.79	2.86	2.65
Cut	18,944	14,547	12,581	86.48	10,268	70.58	4,560	84.47	3,823	70.82	2.96	2.73
Javac	28,794	7,979	6,319	79.19	6,122	76.72	2,115	75.88	2,048	73.48	2.79	2.77
Notepad	50,960	23,898	19,356	80.99	18,702	78.26	7,383	78.74	7,154	76.3	2.77	2.72
Gcc	82,432	70,360	54,559	77.54	46,491	66.07	20,210	75.89	17,828	66.95	2.67	2.52
Grep	85,504	72,811	58,640	80.53	53,707	73.76	20,302	75.27	18,711	69.37	2.76	2.67
Eclipse	86,016	49,769	42,326	85.05	39,722	79.81	14,701	81.55	14,037	77.86	2.82	2.69
Calc	91,408	69,346	45,262	65.26	43,833	63.21	17,623	67.1	17,152	65.3	2.57	2.54
Winmine	96,528	13,375	11,072	82.78	10,530	78.72	4,163	79.9	3,992	76.62	2.76	2.7
<i>Average</i>	<i>61,999</i>	<i>37,239</i>	<i>28,986</i>	<i>77.83</i>	<i>26,458</i>	<i>71.04</i>	<i>10,528</i>	<i>77.76</i>	<i>9,759</i>	<i>71.61</i>	<i>2.77</i>	<i>2.66</i>

* Note: Ri denotes results before chaining and Ri' denotes result after chaining

- R1 = Instructions in valid segments.
- R2 = % Instructions constituting valid segments (CODE) = (R1/TVI) *100
- R3 = Total valid segments.
- R4 = Percent valid segments = (R3/ Total segments)*100
- R5 = Number of segments to which each byte belongs.

The columns for *Total Valid Instructions*, *Instructions*, *Segments* and *Code Inflation* show the values of analysis parameters TVI, RFP, SP and CII respectively. *Note that in case of intermediate-master segment pair we can have one instruction in more than one segment such instruction is counted only once as a part of master segment.*

Table 6-1 depicts that in case of clean executables average TVI is approx. 60% ($Avg. TVI / Avg. File Size$) i.e. if file size is N bytes, $0.60(N)$ bytes may start valid instructions. However, after segmentation actual instructions in the segments is approx 78% of $0.60(N)$. This implies that segmentation further removes approximately 22% of the false positives in case of clean windows executables. The reduction is shown by average value of $R2$ in the table. The results of RFP after chaining depicts that $R2'$ is 71.04 % i.e. a further drop of approx. 7 %.

Fig. 6-3 shows the actual reduction of invalid instructions. We got the highest reduction in instructions for calc.exe for which we got almost 37% reductions. Average reduction for the test runs is 28.96 % (that is the amount of definite data found).

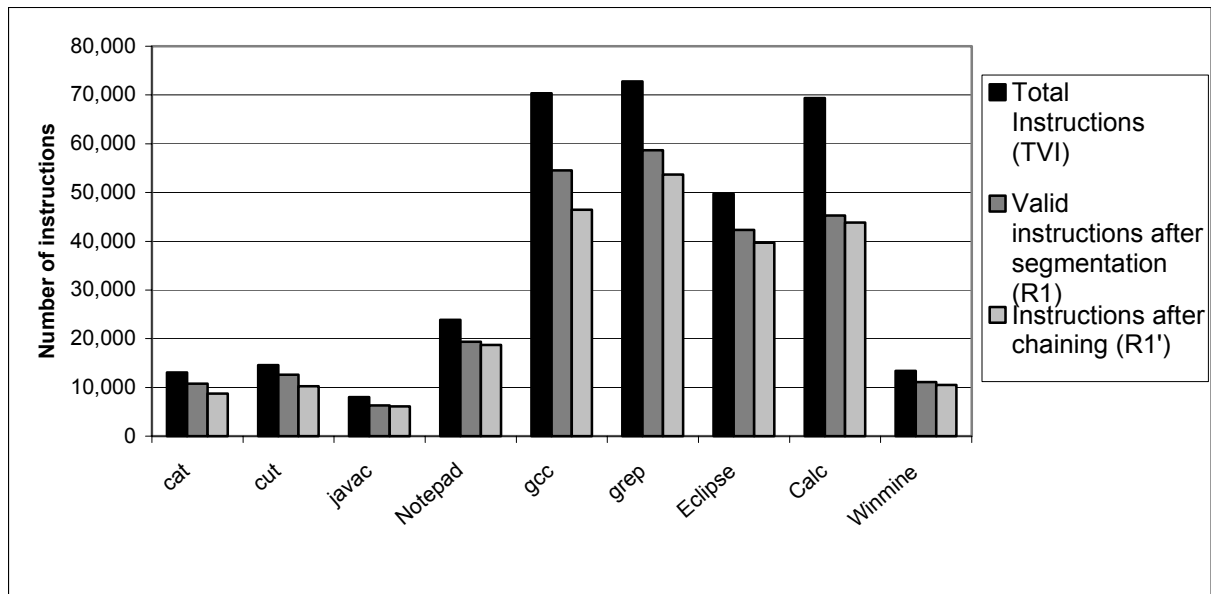


Fig. 6-3 Pruning of invalid instructions: clean executables

Fig. 6-4 depicts the results of SP (Segment Pruning) due to chaining. The results show that approximately 30% of segments on the average are reduced due to segmentation followed by chaining in the “clean” executable.

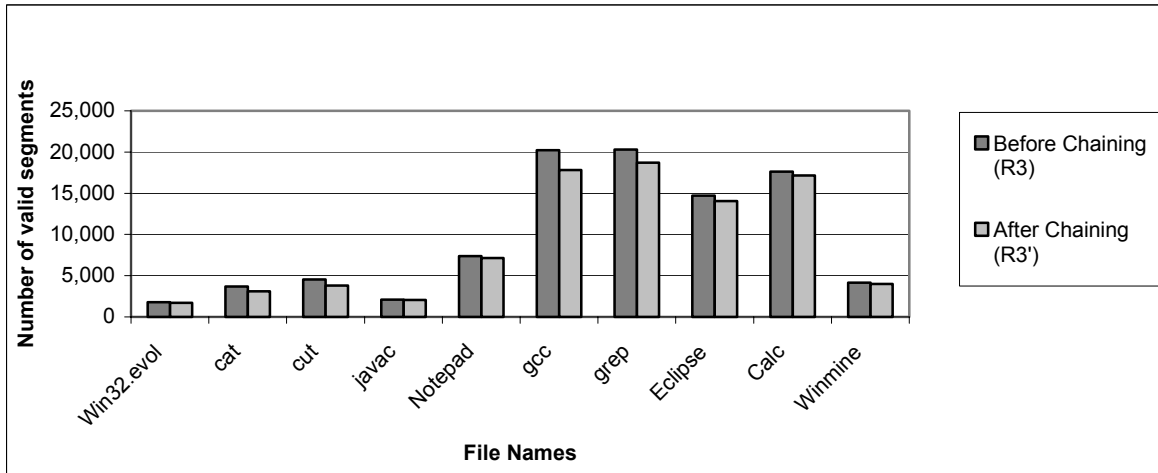


Fig. 6-4 Segment comparisons: clean executables

Fig. 6-5 shows code inflation index of each file. The statistical data presented here show that the average CII after segment chaining is applied is 2.178.

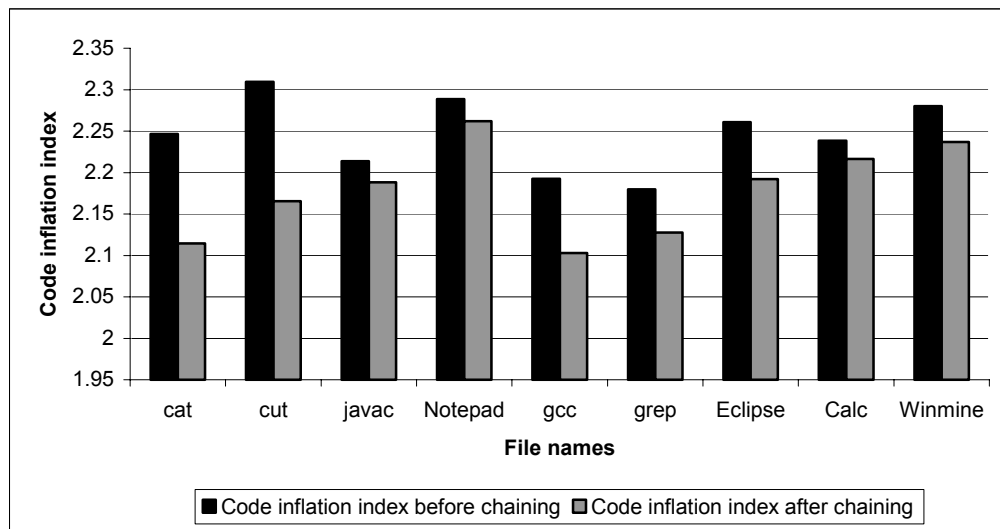


Fig. 6-5. Code inflation index: clean executables

It implies that even after applying segment chaining there are false positives that have to be removed.

6.2.4 Data Analysis: “Malicious” Executable

The analysis parameters were again tested for the following malicious executables in order to compare the results with clean files. The collection of malicious executables includes win32.evol.a, win32.volcano, win32.donut, win32.parvo.a, win32.driller, win32.etap, and win32.apparition. Table 6-2 shows the results for these malicious binaries.

Table 6-2 Results: Malicious Executable (instruction, segments, and overlaps)

Filename (.exe)	File Size (In bytes)	Total Valid Instructions (TVI)	Instructions (RFP)				Segments (SP)				Code Inflation (CII)	
			Before chaining		After chaining		Before chaining		After chaining		Before chaining	After chaining
			R1	R2	R1'	R2'	R3	R4	R3'	R4'	R5	R5'
Evol	12288	5655	4689	82.91	4549	80.44	1788	79.08	1736	76.78	2.126	2.104
vulcano	12288	5199	4181	80.41	3993	76.80	1489	78.53	1411	74.41	2.12	2.11
donut	12800	5480	3034	55.36	3031	55.31	1151	58.27	1148	58.12	1.87	1.87
Zombie	19131	8848	2204	61.52	2006	56	831	63.19	773	58.78	2.15	2.08
parvo	80093	3147	2261	71.84	2250	71.49	893	69.98	886	69.43	2.05	2.05
Zmist	86016	41539	28126	67.7	27031	65.07	10587	65.87	10203	63.48	2.15	2.12
driller	94208	82390	77843	94.48	10386	12.60	4518	71.14	3888	61.22	1.18	1.93
Stepar	150528	14141	11161	82.24	10432	76.87	4061	78.97	3838	74.64	2.22	2.16
etap	160256	39184	33715	86.04	32897	83.95	12618	83.53	12368	81.88	2.36	2.33
Apparition	542861	36583	29786	81.42	24136	65.97	10500	77.49	8911	65.76	2.15	2.01
Average	117046.9	24216	19700	61.14	12071	49.84	4843	72.58	4516	68.43	2.04	2.07

* Note: Ri denotes results before chaining and Ri' denotes result after chaining

- R1 = Instructions in valid segments.
- R2 = % Instructions constituting valid segments (CODE) = (R1/TVI) *100
- R3 = Total valid segments.
- R4 = Percent valid segments = (R3/ Total segments)*100
- R5 = Number of segments to which each byte belongs.

Fig. 6-6 shows the actual reduction of invalid instructions. We got the highest reduction in instructions for *driller.exe* for which the reduction is approximately 87%. Average reduction for the test runs is 50.14% (100% - 49.84%). Approximately 21% increase over results from clean files.

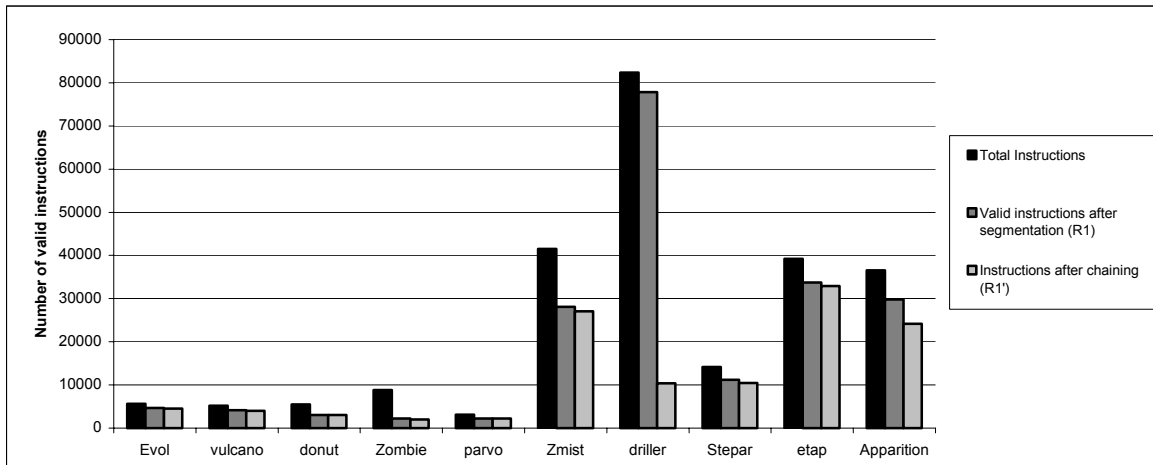


Fig. 6-6 Pruning of invalid instructions: malicious executables

Fig. 6-7 depicts results of SP (Segment Pruning) due to chaining. The results show that approximately 33% of segments are reduced due to segmentation followed by chaining the “malicious” executable.

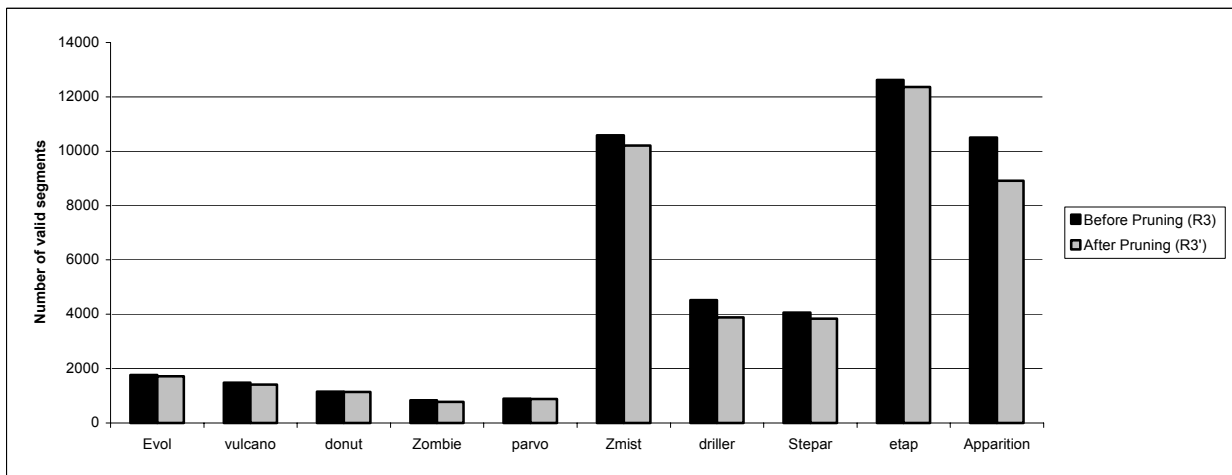


Fig. 6-7 Segment comparisons: malicious executables

Fig. 6-8 shows code inflation index of each file. The statistical data presented here shows that the average CII after segment chaining is applied is 2.07.

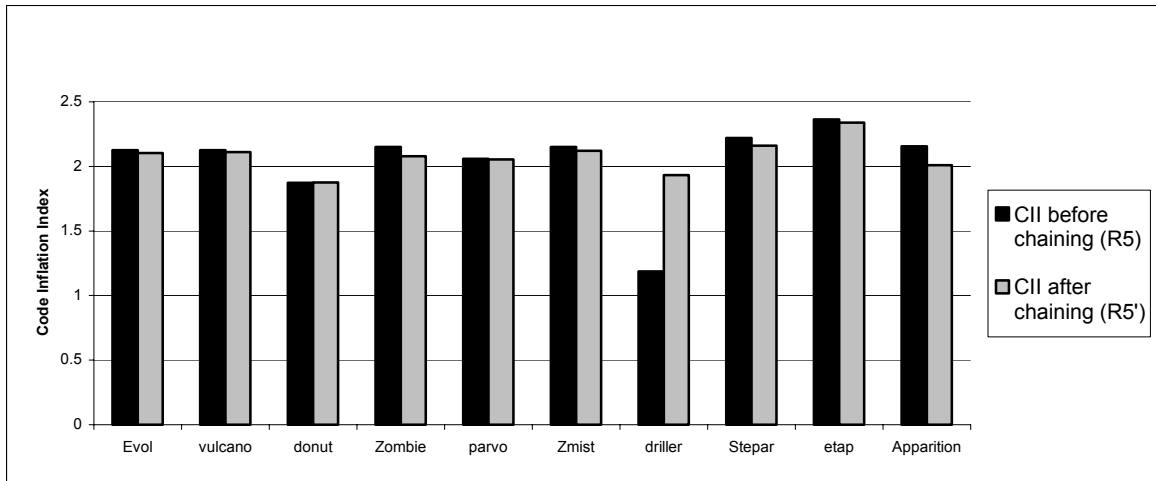


Fig. 6-8 Code inflation index: malicious executables

6.2.5 Data Analysis: Non-Executables

In order to see whether we can classify a binary file in one of the *clean*, *malicious* or *non-executable* category, using just the results from the proposed algorithm, we need data analysis of non-executables as well. The pool of non-executables consists of eight image files and two encrypted files that use data segments as well. The images are further divided as four JPGs and four GIFs (converted from same jpgs) for generality of results.

Table 6-3 Image files used for data analysis.

<i>Tomcat</i>	<i>Jesus</i>	<i>Solar</i>	<i>Navy_Sub</i>
			

Table 6-3 illustrates the images used as a binary input to the disassembler. The tool used to convert JPGs to GIFs is ReaConverterPro [1]. The limitation of the current algorithm is evident from Fig. 6-9 in case of the two compressed executables namely Crypto and Sentinel.

Table 6-4 shows the actual reduction of invalid instructions. We got the highest reduction in instructions for *navy_sub.jpg* for which the reduction is approximately 48%.

The average reduction for the test runs is 45.87%.

Table 6-4 Results: Non-Executable (instruction, segments, and overlaps)

Filename	File Size (In bytes)	TVI	Instructions (RFP)				Segments (SP)				Code Inflation (CII)	
			Before chaining		After chaining		Before chaining		After chaining		Before chaining	After chaining
			R1	R2	R1'	R2'	R3	R4	R3'	R4'	R5	R5'
sentinel A	16384	547	540	98.72	57	10.42	121	97.58	22	17.74	2.50	1.35
solar.jpg	25382	19756	9370	47.42	9070	45.91	3800	49.03	3700	47.74	1.89	1.86
crypto	49152	326	226	69.33	222	68.10	93	69.40	92	68.66	1.86	1.87
Tomcat.jpg	49641	43984	25734	58.50	24764	56.30	10367	58.66	10062	56.94	1.90	1.88
jesus.jpg	67260	60269	34573	57.36	33561	55.68	13784	57.28	13438	55.84	1.92	1.90
navy_sub.jpg	87958	77978	42822	54.91	41276	52.93	17508	55.22	16993	53.60	1.91	1.89
solar.gif	130845	118542	66904	56.43	64877	54.72	26322	56.50	25674	55.11	2.05	2.03
Navy_sub.gif	194923	176784	98272	55.58	95440	53.98	38471	55.60	37543	54.26	2.05	2.03
Tomcat.gif	196819	178524	99634	55.81	96388	54.00	39095	55.79	38132	54.42	2.06	2.04
jesus.gif	244972	223537	125019	55.92	121674	54.43	49102	56.13	48079	54.99	2.09	2.07
Average	106333	90024	50309	55.88	48732	54.13	19866	61.12	19373	51.93	2.02	1.89

* Note: Ri denotes results before chaining and Ri' denotes result after chaining

- R1 = Instructions in valid segments.
- R2 = % Instructions constituting valid segments (CODE) = (R1/TVI) *100
- R3 = Total valid segments.
- R4 = Percent valid segments = (R3/ Total segments)*100
- R5 = Number of segments to which each byte belongs.

Fig. 6-9 depicts results of SP (Segment Pruning) due to chaining. The results show that approximately 49% of segments are reduced due to segmentation followed by chaining the non-executable.

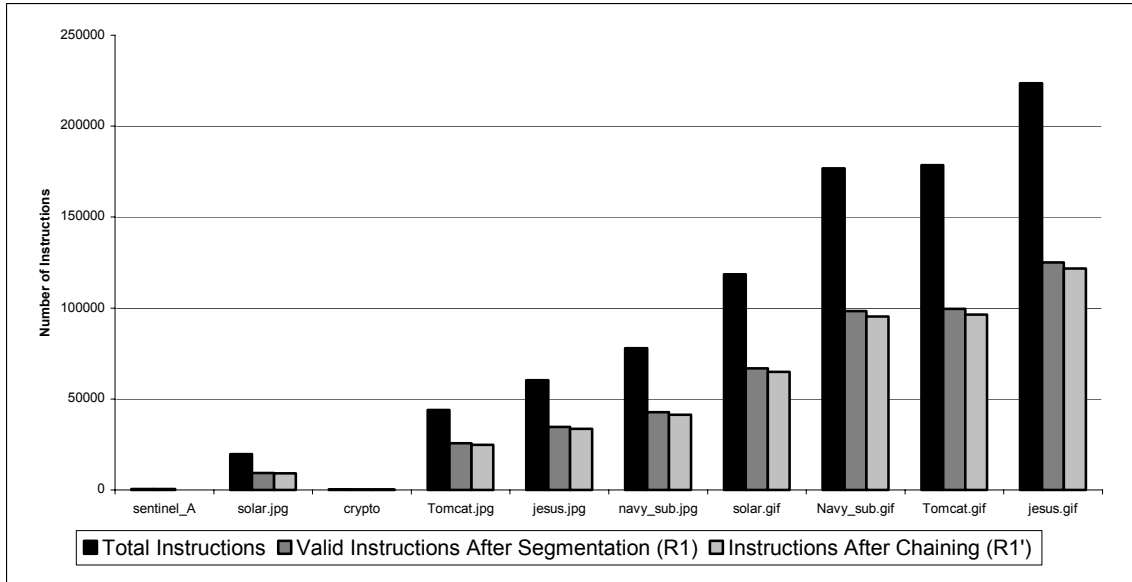


Fig. 6-10 Pruning of invalid instructions: non-executables

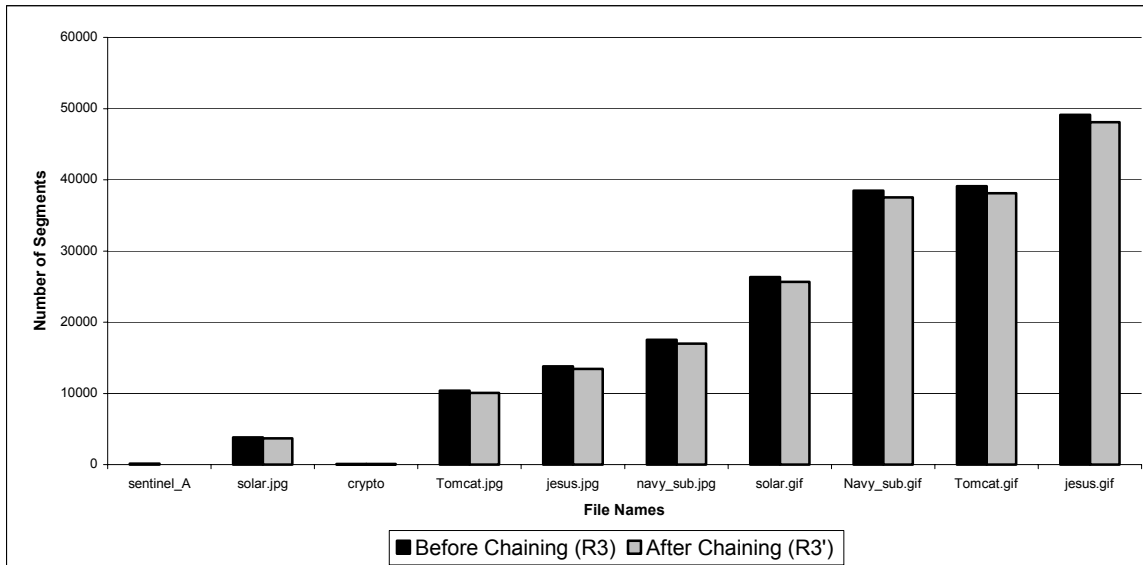


Fig. 6-9 Segment comparisons: non-executables

Fig. 6-11 show code inflation index of each file. The statistical data presented here show that the average CII after segment chaining is applied is 2.07.

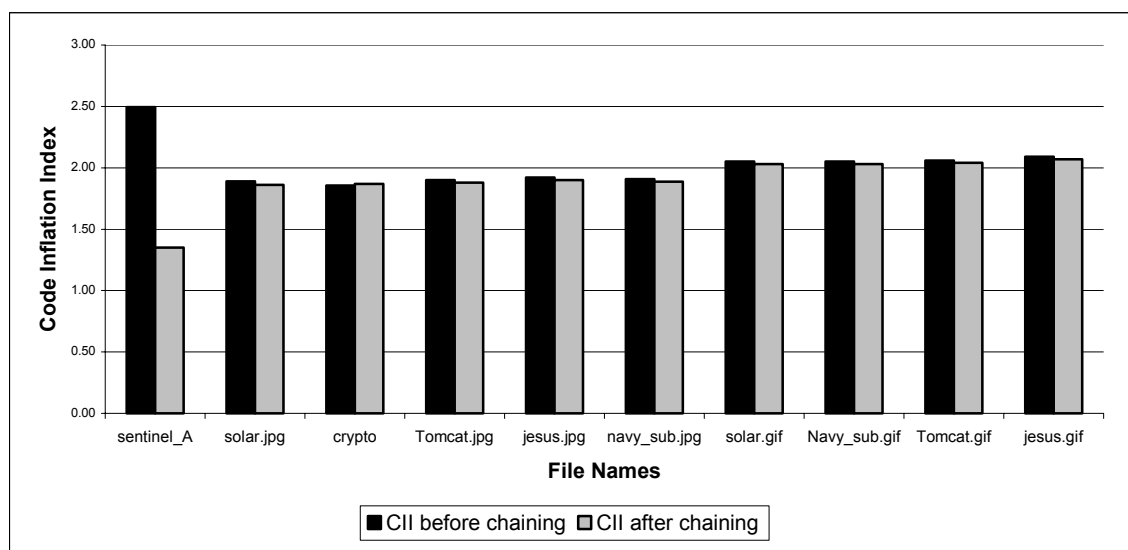


Fig. 6-11 Code inflation index: non-executables

6.2.6 Inference of Data Analysis

Study of data analysis, with respect to analysis parameters, may provide some insight into the feasibility of using the above mentioned statistical approach, in either classifying a binary as clean, malicious or data file or evaluating performance of the suggested algorithms. If we have a file to analyze, differentiating data file from an executable is a simple approach. However, if we were given a buffer of bytes it would be difficult to determine whether it is a buffer of executable bytes or data bytes.

The analysis of clean executables showed that approximately on the average out of every 10 bytes six start a valid instruction i.e. approximately 60%. Whereas in malicious executable only 2 bytes start the instruction out of ten. The results of non-executable data files show that more than 8 bytes start an instruction out of every 10 bytes analyzed. So, can we assume that this is the general trend of all files and we can classify the files on this basis? No, Since there are many factors limiting these results for example if we see results from driller and apparition in Table 6-2, these results vary the final outcome significantly, also the collection of files is too small to statistically prove it. However, there is another significant result worth looking at. It is the percentage

reduction[†] of instructions due to segmentation and chaining. This particular information provides information of the performance of algorithm on certain classes of files. The percentage reduction of instructions in case of clean files came out to be 28.95%, in case of malicious files this value is 45.86%. The significant improvement in reduction of instructions may be due to more sparse code distribution, well connected through jumps, in malicious files. The well-connected yet sparse code is best fit for dividing the program into better-connected segments leading to more reduction through the “ripple effect” described earlier in chapter 5. Thus, the results show that the suggested algorithm is efficient in pruning the invalid instructions and segments from a malicious file leading towards a cleaner disassembly. Finally, the results from data/ non-executables are highly debatable. Nonetheless, the 50% reduction of the instructions due to segmentation and chaining might give an insight of how we can distinguish between data and executable files.

[†] Percentage reduction = $(1 - R1' / TVI) * 100$

7 Related Work

In this section, we briefly describe some reverse engineering techniques (as a representative of flow sensitive and insensitive methods) and show *why* and *how* these are prone to malicious attacks with the intent of thwarting static analysis.

Most obfuscation techniques are based on confusing the two basic algorithms Linear Sweep and Recursive Traversal, by making control flow analysis tougher, inserting junk bytes or delaying inherent property of re-synchronizing of Intel IA-32 architecture after being digressed [17]. Almost all obfuscators and malicious code writers try to attack basic limitations of control flow analysis and data flow analysis. The paper [17] discusses in much detail techniques that can be used to further thwart static analysis by thwarting RT.

In [13] to defeat the static analysis, the authors use techniques to make control-flow analysis of a program dependent on data, they show that this approach will reduce precision of analysis to useless levels. They also prove that the theoretical bound of analysis of programs transformed in such manner is NP complete. We contend that such techniques of attacking flow sensitive methods, if used by hostile programmers, may result in malicious code that is very hard to statically analyze. This is supported by some obfuscation results presented by these obfuscation techniques, wherein [17] shows that the best disassembly tool commercially available fails to disassemble over 65% of the instructions.

The most recent attempt to reduce the indirect jump targets come from [7] who use *values set analysis* to get the *abstract store* of possible jump locations. They remove any address from the set, which are successors of jump but does not start any instruction.

The work we are proposing will help further reduce this abstract set as we are removing whole segments of instructions in such cases.

The closest work to our techniques is the work done by [21]. Their work uses various heuristics to disassemble the binary and separate code and data, but is also dependent upon the knowledge of control flow. The main drawback in the approach to avoid unsafe binary instrumentation is that they mark code as data (i.e. gives false negative) whenever in doubt. This renders an “imprecise” disassembly of the binary file. We say it is closest to our work because it uses a notion of “reset point” that tends to analyze the bytes unmarked (as either code) or data after using control flow analysis. The algorithm scans for a byte that start an instruction from reset point, and as soon as it hits a data byte, it marks all the bytes from reset point till the end byte as data thus, it does not analyze all the bytes for possibly starting an instruction. Thus, this method too results in false negatives. In the next section we show by example how the current algorithms [24] are prone to give false negatives and discuss the new approach in this context.

We recently came across a work [14] almost parallel to the disassembly algorithm we are proposing; the work deals with obfuscated binaries, i.e., it has almost same motive we do to make the disassembler robust against obfuscation and hostile attacks. Their idea of disassembly is close to what we thought i.e., they take this fact that each byte can start an instruction. However, both the works are coming out at the same time, the development of our algorithm is indigenous and independent of the work they have mentioned in their paper. We will briefly discuss the differences and similarities between our approach and the approach [14] is taking.

The approach proposed by [14], follows some assumptions, in that they assume that valid instructions must not overlap and reason that it is extremely hard to find such instructions in real executables. In the segmentation algorithm we are not making this assumption and that leads to more precise disassembly. We can always craft a code that uses bytes in an instruction twice by jumping back into the middle of instruction. This can be a crucial assumption to make in case the hostile programmer decides to obfuscate the code, keeping this in mind. We contend that we are making the fewest assumptions possible to make it more difficult to be exploited by hostile programmer.

Their algorithm starts with dividing whole code into smaller regions called functions. They search for the start of each function through function prologue, where function prologue is the sequence of bytes normally present at function start. Their algorithm searches for all such prologues and divides the code into functions. In our approach we do not divide the executable into functions and disassemble the binary as a whole (we can assume that we are taking whole code as one function). Hostile programmers can change the function prologue by obfuscating it or can insert some junk bytes that look like prologue. This work specifically mentions that, if they get “function start,” that looks like a valid prologue but is not in reality a prologue they might get less accurate results. An obfuscator can easily insert more junk bytes that look like prologue and can increase confusion and resulting in even less accurate results. Once they get functions they disassemble each and every byte of the function as we do for whole program to find all the possible instructions. Since they make assumption that no two valid instructions can be overlapping, they might miss some valid instructions while we are getting all the possible instructions in segmentation algorithm. They divide each

function into basic blocks equivalent to a segment but they cannot have two valid overlapping basic blocks as we have segments, again since they are against the idea of two valid overlapping instructions.

8 Conclusion and Future Work

We have described a novel approach towards disassembly of malicious executables. Our method analyzes each and every byte for possibly starting an instruction, i.e., it does not leave any code behind from disassembly. This approach gives all the possible instructions in the binary executable, thus making it harder for a hostile programmer to hide the malicious program by common obfuscation techniques. Thus the approach may be used with current disassembly techniques to enhance the disassembler capabilities, since this method is more resilient to attacks and based on minimum assumptions.

The proposed method uses a segmentation-based approach wherein it divides the whole code into various overlapping segments giving no false negatives and reducing the false positives. The segmentation-based approach, however, has some limitations that might force it to give some false negatives. It might lead to less precise disassembly, but will not throw the disassembler into complete disarray as is the case with current disassemblers. The main challenge with a segmentation-based disassembler is to remove false positives. The test results of malicious executables show that it is able to reduce up to 87% of instructions (pure data) and up to 46% on the average. The statistics also show that code inflation index on the average is 2.07%; i.e., the valid instructions can reach approximately twice the amount of actual instructions.

In the future work we would like to be able to compare performance of our algorithm with the existing algorithms. We would also be developing more heuristic and statistical approaches to get a unique disassembly instead of multiple paths of disassembly. The tool is being developed based on segmentation algorithm, using java

and eclipse framework. It can be made interactive such that user can decide which path of disassembly to choose and which ones to discard.

Although the segmentation-based approach makes disassembly more resilient to attacks that use common obfuscation techniques like control flow obfuscation and junk byte insertion, it is susceptible to the obfuscations like SEH and self-modifying code.

Attacks on disassembly by obfuscation through SEH or Structured Exception Handling are dangerous, as they obfuscate jump instruction in a way that makes it hard to statically analyze the jump target and, hence, may miss some code while disassembling. The future work might involve development of disassembly techniques / heuristics that counter these attacks.

Attacks to thwart disassembly by obfuscation through code modification at runtime are tougher to counter, since the code changes dynamically as the program runs. We need to develop some methods, as future work, that can deal with such attacks.

Bibliography

- [1] Reasoft Software, "Change Jpg to Gif," <http://www.reasoft.com/rea-jpg-change-gif.shtml>, Last accessed November 16, 2004.
- [2] Datarescue, "Datarescue Sa/Nv, Liege, Belgium, Ida Pro," <http://www.datarescue.com/idabase>, Last accessed November 24, 2004.
- [3] Intel, "Intel Architecture Software Developer's Manual: System Programmer's Guide Volume 3," <http://developer.intel.ru/design/pentium4/manuals/253668.htm>, Last accessed November 16, 2004.
- [4] Swanson Technologies, "Understanding Intel Instruction Sizes," <http://www.swansontec.com/sintel.htm>, Last accessed June 28, 2004.
- [5] VX Heavens, "Virus Creation Tools," <http://vx.netlux.org>, Last accessed June 30, 2004.
- [6] Mi2g, "2004-03-08, News Alert, Economic Damage from Bagle, Mydoom & Netsky 2004-03-30, News Alert, Netsky Climbs to 2nd Worst Malware since 1995," <http://www.mi2g.com/>, Last accessed June 30, 2004.
- [7] G. Balakrishnan and T. Reps, "Analysing Memory Accesses in X86 Executables," in *International Conference on Compiler Construction*, pp. 5-23, 2004.
- [8] C. Batty, "Using the Java Native Interface," <http://www.cs.umanitoba.ca/~eclipse/8-JNI.pdf>, Last accessed June 30, 2004.
- [9] G. Caprino, "Rec - Reverse Engineering Compiler," <http://www.backerstreet.com/rec/rec.htm>, Last accessed June 28, 2004.
- [10] S. Cho, "Win32 Disassembler," <http://www.geocities.com/sangcho>, Last accessed June 30, 2004.
- [11] C. Cifuentes and M. V. Emmerik, "Uqbt: Adaptable Binary Translation at Low Cost," in *IEEE Computer org*, vol. 33, pp. 60-66, 2000.
- [12] E. Gabrilovich and L. Finkelstein, "Jni - C++ Integration Made Easy," http://www.cs.technion.ac.il/~gabr/papers/cuj_jni.pdf, Last accessed November 16, 2004.
- [13] R. N. Horspool and N. Marovac, "An Approach to the Problem of Detranslation of Computer Programs," *The Computer Journal*, vol. 23, pp. 223-229, 1979.

- [14] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static Disassembly of Obfuscated Binaries," in *USENIX Security Symposium*, San Diego, pp. 255-270, 2004.
- [15] A. Lakhota and P. K. Singh, "Challenges in Getting 'Formal' with Viruses," *Virus Bulletin*, pp. 15-19, 2003.
- [16] B. Lee, "Eclipse Project Cdt (C/C++) Plugin Tutorial," Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada February 20, 2004.
- [17] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in *Conference on Computer and Communications Security*, Washington, DC, pp. 290-299, 2003.
- [18] M. Ludvig, "Instruction Format," <http://www.logix.cz/michal/doc/i386/chp17-02.htm#17-02>, Last accessed June 30, 2004.
- [19] N. Mehta and S. Clowes, "A Security Microcosm Attacking and Defending Shiva," <http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-asia-03/bh-asia-03-clowes.pdf>, Last accessed June 30, 2004.
- [20] R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere, "Alto: A Link-Time Optimizer for Compaq Alpha," *Software - Practice and Experience*, vol. 31, pp. 67-101, 2001.
- [21] M. Prasad and T.-c. Chiueh, "A Binary Rewriting Defense against Stack Based Buffer Overflow Attacks," in *USENIX 2003 Annual Technical Conference*, pp. 211-224, 2003.
- [22] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of Executable Code Revisited," in *Ninth Working Conference on Reverse Engineering (WCRE'02)*, Richmond, VA, pp. 45-54, 2002.
- [23] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary Translation," *Communications of the ACM*, vol. 36, pp. 69-81, 1993.
- [24] L. Vinciguerra, L. Wills, N. Kejriwal, P. Martino, and R. Vinciguerra, "An Experimentation Framework for Evaluating Disassembly and Decompilation Tools for C++ and Java," in *10th Working Conference on Reverse Engineering*, pp. 14-23, 2003.
- [25] L. Zeltser, "Reverse Engineering Malware," <http://www.zeltser.com/sans/gcih-practical/revmalw.html>, Last accessed June 30, 2004.

Abstract

Analysis of a binary executable starts with disassembly. Current methods start disassembling a program from an entry point, and continue disassembling either by a simple linear scan or by following the execution paths using static analysis. A hostile programmer can easily fool such methods for disassembly and cause confusion between code and data.

A method for disassembling that does not leave any code behind is proposed. The method does not depend on an entry point; instead, it treats every byte as a potential entry point. It partitions the code into segments where a segment is a largest sequence of instructions terminated by an instruction, such as JUMP, that explicitly modifies the program counter. A segment is invalid if its execution will always lead into data. The method filters out invalid segments. It finally connects all the code segments to give a “precise disassembly.” Results from using the method on ten programs are presented.

We have implemented Binary Disassembler for Malicious Executables (BDME). BDME is a disassembler that is written specifically to target malicious code and obfuscated binaries. For our test data, we are able to reduce up to 87% of false positives while getting no false negatives. BDME is able to find all the potential instructions in the code, thus making it very hard to hide malicious code from analysis through disassembly. The average reduction of 46% of false positives further helps the disassembler in keeping the disassembly precise.

Biographical Sketch

Mr. Aditya Kapoor was born in Bareilly, India on January 5, 1980. He graduated with a degree of Bachelor of Technology, with distinction, in Computer Science and Engineering from the Institute of Engineering and Technology, Chatrapati Sahu Ji Maharaj University (formerly Kanpur University), Kanpur, India in 2001. He entered the master's program in Computer Science at the University of Louisiana at Lafayette in Fall 2002. Following completion of this degree, he will pursue a Ph.D. in the area of software security.