

CS699/IT803
Professor Yutao Zhong

**Static Analysis of Executables to
Detect Malicious Patterns**
by Mihai Christodorescu and Somesh Jha

Chien-Chih Lin
clin3@gmu.edu
March 30, 2006

Outline

- Introduction
- Motivation
- Background
- Proposed Approach
- Implementation
- Evaluation Methodology
- Results
- Future Direction and Follow Up
- Conclusion

Introduction

- Ensuring the **continuity of business operation** is a crucial issue in the information age.
- A well-designed system should develop a **security component** to protect its intellectual asset.
- Meanwhile, a malicious code self-replicates across a network to **exploit the vulnerability** and **cause the damage** of systems.
- In defending malicious codes, malicious code detection is an **obfuscation-deobfuscation game** between malicious code writers and researchers.

Introduction

- Therefore, this paper presents an **architecture for detecting malicious patterns** in executables that is resilient to common obfuscation transformations.
- This paper tests **three commercial virus scanners** against code-obfuscation attacks and as a result, the three commercial virus scanners could be **subverted by very simple obfuscation transformations**.

Motivation

- Currently, malicious code detection is an obfuscation-deobfuscation game.
- In this obfuscation-deobfuscation game, the authors want to present **a static analysis framework** for detecting malicious code patterns in executables to against common obfuscation transformations.

Background

- Category of Malicious codes
- Current Defense Strategy
- Virus signatures
- Obfuscation Techniques

Background – Category of Malicious codes

Malicious code is usually classified according to its propagation method and goal into the following categories:

- **viruses** are programs that self-replicate within a host by attaching themselves to programs and/or documents that become carriers of the malicious code;
- **worms** self-replicate across a network;
- **trojan horses** masquerade as useful programs, but contain malicious code to attack the system or leak data;
- **back doors** open the system to external entities by subverting the local security policies to allow remote access and control over a network;
- **spyware** is a useful software package that also transmits private user data to an external entity.

Combining two or more of these malicious code categories can lead to powerful attack tools.

Background – Category of Malicious codes

A **polymorphic virus** uses multiple techniques to prevent signature matching.

- nop-insertion
- code transposition (changing the order of instructions and placing jump instructions to maintain the original semantics)
- register reassignment (permuting the register allocation).

Metamorphic viruses attempt to evade heuristic detection techniques by using more complex obfuscations.

- code transposition
- substitution of equivalent
- instruction sequences
- register reassignment
- entry point obscuring (EPO) viruses

Background –

Current Defense Strategy

According to McGraw and Morrisett, a system can use four principle approaches to defend itself when a malicious code arrives.

- **Analyzing** the code and reject it if there is the potential that executing it will cause harm
- **Rewriting** the code before executing it so that it can do no harm
- **Monitoring** the code while its executing and stop it before it does harm, or
- **Auditing** the code during executing and take policing action if it did some harm.

Background –

Current Defense Strategy

By applying these approaches, current strategy in malicious code analysis includes

- **Operating System (OS) based Reference Monitors** to enforce the monitor of address translation hardware, distinct supervisor and user modes, timer interrupts, and system calls by the computer hardware and operating system,
- **Scanning for know Malicious Code** by applying a black listing strategy to compare the know “signature” in most commercial anti-virus products, and
- **Code Signing** used a private key to sign code, both ensuring transmission integrity and enabling policy defined by trust in the signer.

Background – Virus Signatures

The classic virus-detection techniques look for the presence of a virus-specific sequence of instructions (called a **virus signature**) inside the program.

- For example, the Chernobyl/CIH virus is detected by checking for the hexadecimal sequence:
E800 0000 005B 8D4B 4251 5050
0F01 4C24 FE5B 83C3 1CFA 8B2B
- This corresponds to the following IA-32 instruction sequence, which constitutes part of the virus body:
E8 00 00 00 00 call 0h
5B pop ebx
8D 4B 42 lea ecx, [ebx + 42h]
51 push ecx
50 push eax
50 push eax
0F 01 4C 24 FE sidt [esp - 02h]
5B pop ebx
83 C3 1C add ebx, 1Ch
FA cli
8B 2B mov ebp, [ebx]

This classic detection approach is effective when the virus code **does not change** significantly over time.

Background – Obfuscation Techniques

The four common obfuscation transformations: dead-code insertion, code transposition, register reassignment, and instruction substitution.

- **Dead-code insertion** adds a sequence of “nop” instructions to modify a program without changing its own behavior, known as “trash insertion.”
- **Code transposition** changes the order of binary image that is different from the execution order.
- **Register reassignment** exchanges usage of one register with other in a certain range.
- **Instruction substitution** uses a set of equivalent instruction sequences to replace another instruction sequence.

Background – Obfuscation Techniques

<i>Original code</i>	<i>Code obfuscated through dead-code insertion</i>	<i>Code obfuscated through code transposition</i>	<i>Code obfuscated through instruction substitution</i>
call 0h	call 0h	call 0h	call 0h
pop ebx	pop ebx	pop ebx	pop ebx
lea ecx, [ebx+42h]	lea ecx, [ebx+42h]	jmp S2	lea ecx, [ebx+42h]
push ecx	nop (*)	S3: push eax	sub esp, 03h
push eax	nop (*)	push eax	sidt [esp - 02h]
push eax	push ecx	sidt [esp - 02h]	add [esp], 1Ch
sidt [esp - 02h]	push eax	jmp S4	mov ebx, [esp]
pop ebx	inc eax (**)	add ebx, 1Ch	inc esp
add ebx, 1Ch	push eax	jmp S6	cld
cld	dec [esp - 0h] (**)	S2: lea ecx, [ebx+42h]	mov ebp, [ebx]
mov ebp, [ebx]	dec eax (**)	push ecx	
	sidt [esp - 02h]	jmp S3	
	pop ebx	S4: pop ebx	
	add ebx, 1Ch	cld	
	cld	jmp S5	
	mov ebp, [ebx]	S5: mov ebp, [ebx]	

Figure 3: Examples of obfuscation through dead-code insertion, code transposition, and instruction substitution. Newly added instructions are highlighted.

Proposed Approach

A Virus Signature-Based Approach in Virus Detection

- A virus V with a known signature σ and an obfuscated version $O(V)$.
- Finding an σ' semantically equivalent to σ
(Based on a recent result that shows that a computationally bound adversary cannot obfuscate a virus to completely hide its malicious behavior)
- A Proposed Semantic Analysis
- (1) Malicious code automaton (MCA) is a generalization of the finite state automaton.
- (2) The CFG is an annotated control flow graph from a program.
- (3) Defines two Lists, $L(\text{pre})$ based on the MCA and $L(\text{post})$ based on the CFG
- (4) Examines $L(\text{pre}) \cap L(\text{post})$;
 - If $L(\text{pre}) \cap L(\text{post}) = \varphi$ is free of the virus
 - If $L(\text{pre}) \cap L(\text{post}) \neq \varphi$ is not free of the virus

Implementation

- A prototype tool, SAFE (a static analyzer for executables) for detecting malicious patterns in x86 executables

Implementation – Architecture

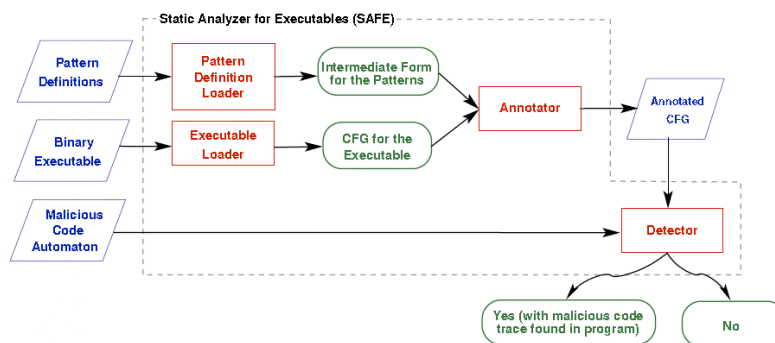


Figure 5: Architecture of the static analyzer for executables (SAFE).

Implementation – Executable Loader

- **The executable loader** uses two off-the-shelf components, *IDA Pro* and *CodeSurfer*. *IDA Pro* (by DataRescue) is a commercial interactive disassembler. *CodeSurfer* (by GrammaTech, Inc.) is a program-understanding tool that performs a variety of static analyses.
- *CodeSurfer* provides an API for access to various structures, such as the CFGs and the call graph, and to results of a variety of static analyses, such as points-to analysis.

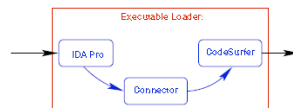
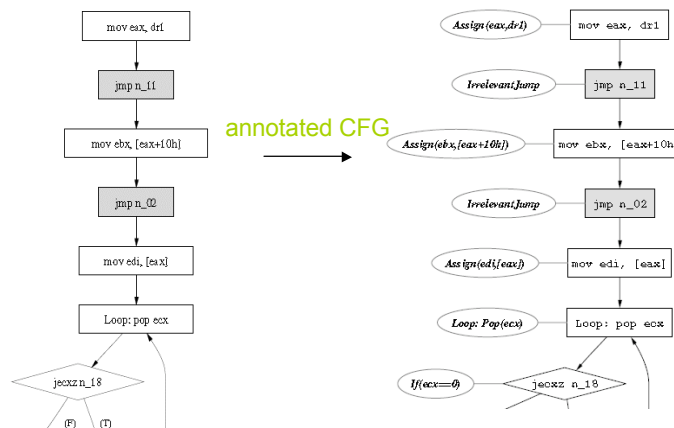


Figure 6: Implementation of executable loader module.

Implementation – Annotator

- **The annotator** is a component that inputs a CFG from the executable and the set of abstraction patterns and produces an annotated CFG, the abstract representation of a program procedure.



Implementation – Detector

- **The detector** is a component computes whether the malicious code (represented by the malicious code automaton) appears in the abstract representation of the executable (created by the annotator).

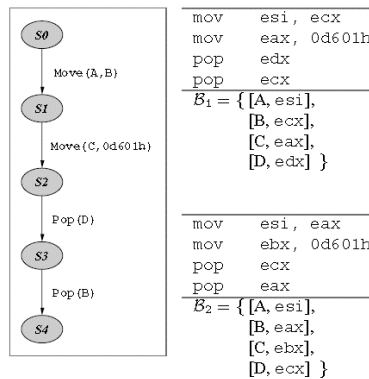


Figure 10: Malicious code automaton for a Chernobyl virus code fragment, and instantiations with different register assignments, shown with their respective bindings.

Implementation – Detector

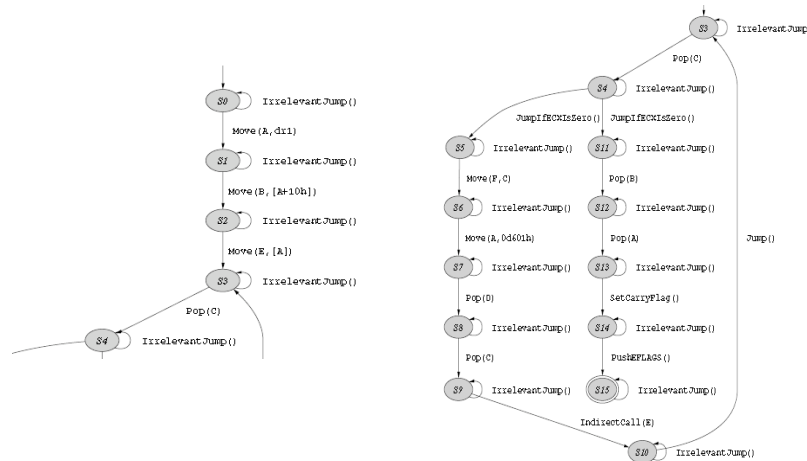


Figure 11: Malicious code automaton corresponding to code fragment from Figure 6.

Implementation – Detector

Input: A list of patterns $\Sigma = \{P_1, \dots, P_r\}$, a malicious code automaton $\mathcal{A} = (V, \Sigma, S, \delta, S_0, F)$, and an annotated CFG $P_\Sigma = \langle N, E \rangle$

Output: *true* if the program is likely infected, *false* otherwise

MALICIOUSCODECHECKING($\Sigma, \mathcal{A}, P_\Sigma$)

```

(1)  $L_{n_0}^{pre} \leftarrow \{[s, \emptyset] \mid s \in S_0\}$ , where  $n_0 \in N$  is the entry node of  $P_\Sigma$ 
(2) foreach  $n \in N$  do  $L_n^{pre} \leftarrow \emptyset$ 
(3) foreach  $n \in N$  do  $L_n^{post} \leftarrow \emptyset$ 
(4)  $WS \leftarrow \emptyset$ 
(5) do
(6)    $WS_{old} \leftarrow WS$ 
(7)    $WS \leftarrow \emptyset$ 
(8)   foreach  $n \in N$ 
(9)     if  $L_n^{pre} \neq \bigcup_{m \in P_{previous}(n)} L_m^{post}$  // update pre information
(10)       $L_n^{pre} \leftarrow \bigcup_{m \in P_{previous}(n)} L_m^{post}$ 
(11)       $WS \leftarrow WS \cup \{n\}$ 
(12)   foreach  $n \in N$  // update post information
(13)      $NewL_n^{post} \leftarrow \emptyset$ 
(14)     foreach  $[s, B_s] \in L_n^{pre}$ 
(15)       foreach  $[\Gamma, \mathcal{B}] \in Annotation(n)$  // follow a transition
(16)          $\wedge Compatible(B_s, \mathcal{B})$ 
(17)         add  $[\delta(s, \Gamma), \mathcal{B}_s \cup \mathcal{B}]$  to  $NewL_n^{post}$ 
(18)     if  $L_n^{post} \neq NewL_n^{post}$ 
(19)        $L_n^{post} \leftarrow NewL_n^{post}$ 
(20)        $WS \leftarrow WS \cup \{n\}$ 
(21) until  $WS = \emptyset$ 
(22) return  $\exists n \in N . \exists [s, B_s] \in L_n^{post} . s \in F$ 

```

Implementation – Detection Capability

- The detection tool can handle:
 - ✓ NOP-insertion
 - ✓ Code reordering (irrelevant jumps and branches)
 - ✓ Register renaming
- Work in progress to detect:
 - Malicious code split across procedures (need inter-procedural analysis)
 - Obfuscations using complex data structures (need integration with pointer analyses)

Evaluation Methodology

- The major goal of experiments were to **measure the execution time of SAFE** and **find the false positive and negative rates**.
 - Ten obfuscated versions of the four viruses and **four** MCAs from four viruses.
 - Four Viruses**
 - Chernobyl
 - z0mbie-6.b
 - f0sf0r0
 - Hare
 - The testing environment consisted of a Microsoft Windows 2000 machine.
 - The hardware configuration included an AMD Athlon 1 GHz processor and 1 GB of RAM.
 - We used CodeSurfer version 1.5 patchlevel 0 and IDA Pro version 4.1.7.600.
- Four Commercial and **Benign programs**
- tiffdither.exe* **9k**
 - winmine.exe* **96k**
 - spyxx.exe* **500k**
 - QuickTimePlayer.exe* **1M**

Results –

Testing Commercial Anti-Virus Products

		Norton® Antivirus 7.0	McAfee® VirusScan 6.01	Command® Antivirus 4.61.2	SAFE
Chernobyl	original	✓	✓	✓	✓
	obfuscated	✗ ^[1]	✗ ^[1,2]	✗ ^[1,2]	✓
z0mbie-6.b	original	✓	✓	✓	✓
	obfuscated	✗ ^[1,2]	✗ ^[1,2]	✗ ^[1,2]	✓
f0sf0r0	original	✓	✓	✓	✓
	obfuscated	✗ ^[1,2]	✗ ^[1,2]	✗ ^[1,2]	✓
Hare	original	✓	✓	✓	✓
	obfuscated	✗ ^[1,2]	✗ ^[1,2]	✗ ^[1,2]	✓

Obfuscations considered: ^[1] = nop-insertion (a form of dead-code insertion)
^[2] = code transposition

Table 1: Results of testing various virus scanners on obfuscated viruses.

Results

- **Testing on malicious code:**
SAFE's false positive and negative rate were 0.
- **Testing on benign code:**
SAFE detector reported "negative" in each case, i.e., the false positive rate is 0.

Results –

Performance in Execution Time (Malicious Codes)

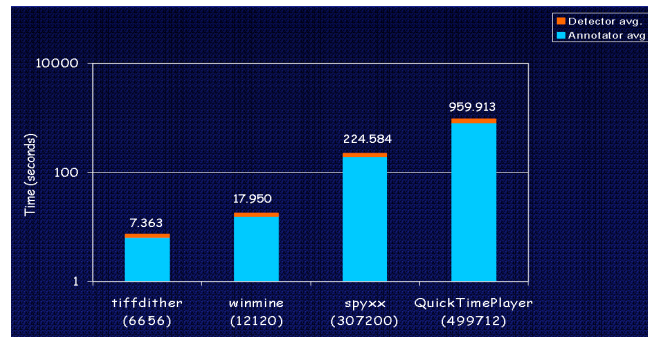
	Annotator		Detector	
	avg.	(std. dev.)	avg.	(std. dev.)
Chernobyl	1.444 s	(0.497 s)	0.535 s	(0.043 s)
z0mbie-6.b	4.600 s	(2.059 s)	1.149 s	(0.041 s)
f0sf0r0	4.900 s	(2.844 s)	0.923 s	(0.192 s)
Hare	9.142 s	(1.551 s)	1.604 s	(0.104 s)

Table 5: SAFE performance when checking obfuscated viruses for false negatives.

	Annotator		Detector	
	avg.	(std. dev.)	avg.	(std. dev.)
z0mbie-6.b	3.400 s	(1.428 s)	1.400 s	(0.420 s)
f0sf0r0	4.900 s	(1.136 s)	0.840 s	(0.082 s)
Hare	1.000 s	(0.000 s)	0.220 s	(0.019 s)

Table 6: SAFE performance when checking obfuscated viruses for false positives against the Chernobyl/CIH virus.

Results – Performance in Execution Time (Benign Codes)



	Executable size	.text size	Procedure count	Annotator		Detector	
				avg.	(std. dev.)	avg.	(std. dev.)
tiffdither.exe	9,216 B	6,656 B	29	6.333 s	(0.471 s)	1.030 s	(0.043 s)
winmine.exe	96,528 B	12,120 B	85	15.667 s	(1.700 s)	2.283 s	(0.131 s)
spyxx.exe	499,768 B	307,200 B	1,765	193.667 s	(11.557 s)	30.917 s	(6.625 s)
QuickTimePlayer.exe	1,043,968 B	499,712 B	4,767	799.333 s	(5.437 s)	160.580 s	(4.455 s)

Table 7: SAFE performance in seconds when checking clean programs against the Chernobyl/CIH virus.

Future Direction in 2003

- **New languages**
 - Scripts: Visual Basic (in progress), ASP, JavaScript
 - Multi-language malicious code
- **Attack diversity**
 - Beyond virus patterns: worms, trojans
- **Irrelevant sequence detection**
 - Decision procedures
 - Theorem provers

Follow Up in 2004

Testing Malware Detectors
Mihai Christodorescu and Somesh Jha

Malware name	Malware detector	Extracted virus signature
Anna Kournikova	Norton AntiVirus	Execute 67iqom5JE4z("X)udQ0VpgjnH...7042")
	Sophos Antivirus	1 Execute 67iqom5JE4z("X)udQ0VpgjnH...7042")
		5 StTP1MoJ3ZU = Mid(hPesaKrcocjS, 1, 1)
		6 WHz23rBqlo7 = Mid(hPesaKrcocjS, 1 + 1, 1)
		8 StTP1MoJ3ZU = Chr(10)
		10 StTP1MoJ3ZU = Chr(13)
		12 StTP1MoJ3ZU = Chr(32)
		14 StTP1MoJ3ZU = Chr(Asc(StTP1MoJ3ZU) - 2)
		16 WHz23rBqlo7 = Chr(10)
	18 WHz23rBqlo7 = Chr(13)	
20 WHz23rBqlo7 = Chr(32)		
22 WHz23rBqlo7 = Chr(Asc(WHz23rBqlo7) - 2)		
24 67iqom5JE4z = 67iqom5JE4 & WHz23rBqlo7 & StTP1MoJ3ZU		
McAfee Virus Scan	The whole body of the malware.	
Melissa	Norton AntiVirus	The whole body of the malware.
	Sophos Antivirus	The whole body of the malware.
	McAfee Virus Scan	83 statements from the malware body.
Lucky2	Norton AntiVirus	F80.CopyFile Melhacker, target.Name, 1
	Sophos Antivirus	The whole body of the malware.
Yovp	McAfee Virus Scan	1 Dim Melhacker, WshShell, F80, Vx, VirusLink
		6 Melhacker = Wscript.ScriptFullName
		7 Vx = Left(Melhacker, InStrRev(Melhacker, "\"))
	Norton AntiVirus	8 F80.CopyFile Melhacker, target.Name, 1
Sophos Antivirus	9 dosfile.writeline("command /t /c copy C:\virus.vbs A:\\")	
	10 dosfile.writeline("del C:\virus.vbs")	
McAfee Virus Scan	1 On Error Resume Next	
	2 Dim feo, wsh, dosfile, openvir, copyov	
	3 Set feo = createobject("scripting.filesystemobject")	
	6 Set dosfile = feo.createtextfile("c:\dosfile.bat", true)	
	7 dosfile.writeline("echo off")	
8 dosfile.writeline("cd %windir%")		

Table 3: Signatures discovered by our signature-extraction algorithm. Where relevant, line numbers for each statement are provided.

Follow Up in 2005

Semantics-Aware Malware Detection
Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant

Malware family	Template detection		Running time	
	Decryp-tion loop	Mass-mailer	Avg.	Std. dev.
Netsky	100%	100%	99.57 s	41.01 s
B[e]agle	100%	100%	56.41 s	40.72 s
Sober	100%	0%	100.12 s	45.00 s

Table 4. Malware detection using algorithm \mathcal{A}_{MD} for 21 e-mail worm instances.

Obfuscation type	Algorithm \mathcal{A}_{MD}		McAfee VirusScan
	Average time	Detection rate	
Nop	74.81 s	100%	75.0%
Stack op	159.10 s	100%	25.0%
Math op	186.50 s	95%	5.0%

Table 5. Evaluation of algorithm \mathcal{A}_{MD} on a set of obfuscated variants of B[e]agle.Y. For comparison, we include the detection rate of McAfee VirusScan.

Question

?
???
?????
????????
?????????
????????????
?????????????
????????????????

Conclusion

- The rapid growth of malicious codes increases the importance of detecting malicious and vulnerability of systems in computer security.
- Static analysis provides a solution of defending malicious codes.
- As a result, static analysis has better performance in detecting malicious codes.
- The Challenge of Virus Detection in novel virus (virus without any discovered signature) and virus family (different version of the same virus)
- ??? Change of CodeSufer (program-understanding tool) will change the performance better or worse???

Acknowledgement

- Thanks for Professor Zhong for her feedback.
- Thanks for Mihai and Somesh for their powerpoints of this paper.
- Thanks for Dan for his summary and comment.
- Thanks for the support from my wife and supervisor.