Examining Self-Modifying Code

Drew Ivarson, Union College CS Department
Advisors: Prof. Anderson, Prof. Spinelli
Overview

- Background
- Motivation
- My contributions
Examining Self-Modifying Code

- What code am I talking about?
- How do I examine it?
- How is it self-modifying?
Examining Self-Modifying Code

- The Code found in executable files

  Binary = Assembly instructions

  01010001020101 = INC 0x1
  0200        = MOVB 0x1 0x2
                 INC 0x1
                 JMP 0X0
Examining Self-Modifying Code

- **Dynamic Analysis**
  - run the program
  - evaluate the results of each instruction being executed

- **Static Analysis**
  - not running the program!
  - quickly cover all possible traversals
Examining Self-Modifying Code

non-self-modifying:

0x0: movb 0x7 reg1
0x3: inc reg1
0x5: jmp 0x0
0x7: inc reg1
Self-Modifying?

non-self-modifying:

0x0: movb 0x7  \textcolor{yellow}{\text{reg1}}
0x3: inc reg1
0x5: jmp 0x0
0x7: inc reg1

self-modifying:

0x0: movb 0x7 0x6
0x3: inc reg1
0x5: jmp 0x0  \text{\textcolor{black}{\text{\textup{\textrightarrow}}}}  \text{jmp 0x7}
0x7: inc reg1
Examining Self-Modifying Code:

1. Binary Files - binary (assembly) code
2. Static Analysis - not running it
3. Self-Modifying: writing to instruction memory instead of data memory
1260, a self-modifying virus

Before Running:
- Do register math
- Read from memory

While Running:
- Do register math
- Read from memory
- SEND PERSONAL INFORMATION TO SOME IP

After Running:
- Same as before...
A Model for Self-Modifying Code

- Answer to static analysis problem
- AMB algorithm and data structure
- A *model*, so it hasn’t been implemented!
Input and Output to AMB

Binary File:
0001001010101010
01010101010101010
101010111111110
101010100010101
101010101010101

AMB

Control Flow Graph
Control Flow Graphs (CFGs)

- Graph to show program control flow, ie, function calls, conditional statements
- A picture of a traversal through a program
while (true)
    if (Drew.has_goldfish())
        eat_handful();
    else
        cry();
        back_to_work();
A more assembled example

0x0: movb 0xb 0x6
0x3: inc reg1
0x5: jmp 0x7
0x7: inc reg2
0x9: dec reg1
0xb: movb 0x10 0x6
0xe: jmp 0x5
0x10: end

Self-modification!!
A CFG of our new example

0x0: movb 0xb 0x6
0x3: inc reg1
0x5: jmp 0x7
0x7: inc reg2
0x9: dec reg1
0xb: movb 0x10 0x6
0xe: jmp 0x5
0x10: end
AMB Algorithm

- Conservative Estimate

while (state of instruction memory is changing)
    recurse over the program given the current state of memory
    store results of instructions that write to memory, and the results of instructions that change the control flow
Summary of The Model

Specific Instructions:
- 0x0: movb 0xb 0x6
- 0x3: inc reg1
- 0x5: jmp 0x7
- 0x7: inc reg2
- 0x9: dec reg1
- 0xb: movb 0x10 0x6
- 0xe: jmp 0x5
- 0x10: end

AMB Algorithm:
- CodeBytes
- Instructions
My Contribution

- Implement this algorithm
- Bring it from a model to reality
  - User-defined instruction sets
  - User-written test programs
  - Graphical output
User-Defined Instruction Sets

- Abstract Syntax
  - Writes
  - Gotos
  - Skips

Example:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Length</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>3</td>
<td>MOVB</td>
</tr>
<tr>
<td>01</td>
<td>2</td>
<td>INC</td>
</tr>
<tr>
<td>02</td>
<td>2</td>
<td>JMP</td>
</tr>
</tbody>
</table>

 Opcode, length, name, abstract syntax
Implementation

- **State-Enhanced Control Flow Graph**
  - 0x0: INC 0x1
  - 0x2: MOVB 0x2 0x1
  - 0x5: INC 0x1

- **Binary File**
  - Parse
  - Instructions: inc 0x1, movb 0x2 0x1, inc 0x1

- **AMB Analysis**

- **State-Variant Data Structures**
  - CodeBytes, all possible values of each memory address:
    - 0x0: 0x1
    - 0x1: 0x1, 0x2
    - 0x2: 0x0
    - 0x3: 0x2
    - 0x4: 0x1
    - 0x5: 0x1
    - 0x6: 0x1
  - InstructionBytes, all possible instructions with all of their possible control flow targets:
    - 0x0: inc 0x1, inc 0x2
    - targets: 0x2
    - 0x2: movb 0x2 0x1
    - targets: 0x2
    - 0x5: inc 0x1
Results

Simple, no modification program:

0x0: INC 0x1

0x2: INC 0x1

0x4: INC 0x1

Simple, self-modifying program:

0x0: INC 0x1

0x2: MOV 0x2 0x1

0x5: INC 0x1

0x0: INC 0x1

INC 0x2

Already an impossible edge!
This is a 10 line program with no jumps!

Algorithm computes over a million edges!
Before running the algorithm, VIRUS looks like an unreachable line.
Conclusion

- Some optimization required
  - Too many edges and nodes
  - Remove unreachable code
- Detected VIRUS
- Generated graphs based on user-defined instruction sets and user-written programs
- Did not conquer polymorphic code engines
Future Work

- Expand to full instruction sets (like an actual assembly language)
- Top priority: algorithm optimization