Senior Project – Computer Science – 2015

Examining Self-Modifying Code

Drew Ivarson

Advisors – Prof. Anderson, Prof. Spinelli

Introduction

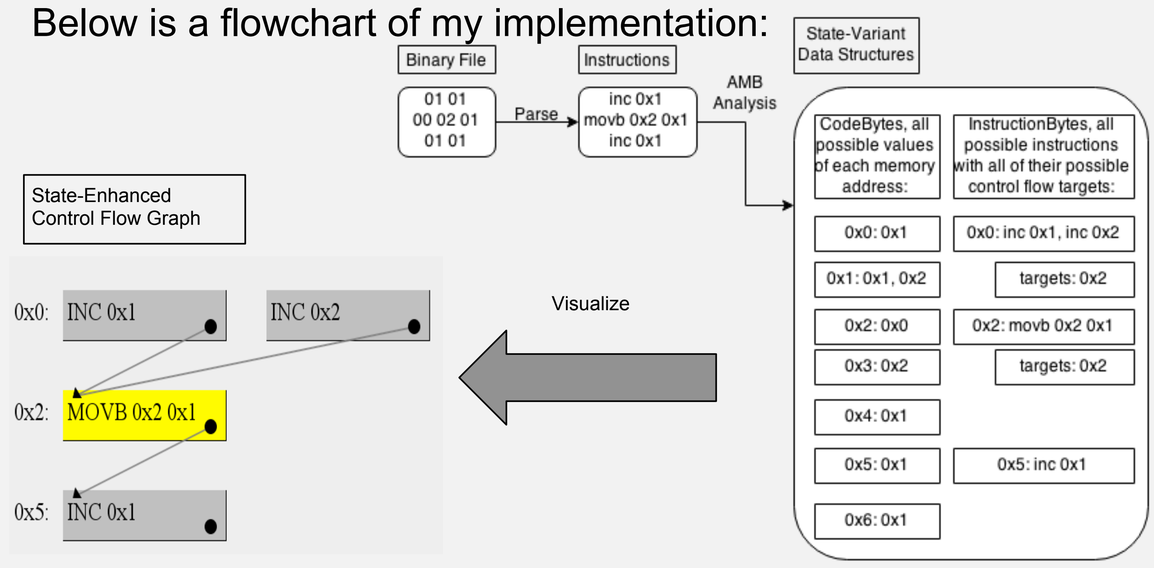
Self-Modifying code is used for both good and bad. As a means of hiding program internals, software companies use it to prevent hackers from “cracking” the authentication on their software. It is also used in malware to avoid being detected by malware analysts and anti-virus software. This is accomplished by writing to its own instruction memory space. Its power comes from being difficult to analyze, because traditional analysis tools don’t account for changing instruction space. In this project, we implement a model proposed by Anckaert et al. [1] to create an analysis tool specifically for tracking self-modification. This model creates a control flow graph of a conservative estimate of all possible execution paths of a binary file. We implement this model with the goal of identifying its strengths and weaknesses and created a test environment. Our visualizations show both positive and negative aspects of this model.

Implementation

To implement the algorithm, my software has 3 steps:

1) Parse Input
2) Analyze Input (with AMB)
3) Visualize the Results of the Analysis

Below is a flowchart of my implementation:



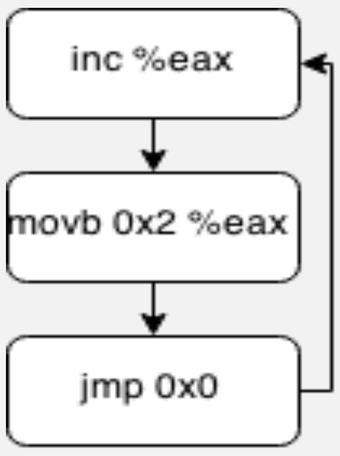
Background

The following examples demonstrate our definition of self-modifying code

1: Non-Self-Modifying	2: Self-Modifying
0x0: inc %eax	0x0: inc %eax
0x2: movb 0x2 %eax	0x2: movb 0x2 0x0
0x5: jmp 0x0	0x5: jmp 0x0

Note the difference between the underlined portions. In 1, the movb instruction sets the value 0x2 to the register %eax. 2, on the other hand, sets the value 0x2 to the memory address 0x0. 0x0 is the address of the first instruction, so 2 is modifying the address space.

Figure 1:



In Figure 1, we have a control flow graph which restates example 1. This is a traditional control flow graph, which does not account for varying instruction memory. To represent example 2, we need a new type of control flow graph.

AMB is the algorithm proposed by Anckaert et al. [1] to generate a *state-enhanced* control flow graph, an SE-CFG. An SE-CFG is a control flow graph which also tracks the state of instruction memory.

AMB Algorithm:

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

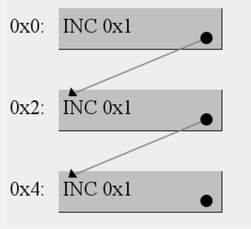
References:

1: Bertrand Anckaert, Matias Madou, and Koen De Bosschere. 2006. A model for self-modifying code. In Proceedings of the 8th international conference on Information hiding (IH'06), Jan L. amenisch, Christian S. Collberg, Neil F. Johnson, and Phil Sallee (Eds.). Springer-Verlag, Berlin, Heidelberg, 232-248.

Results and Evaluation

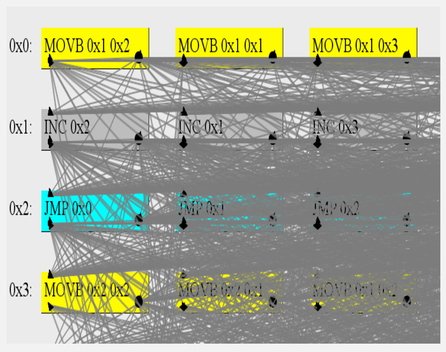
As shown in the implementation section, we have created visualizations based on the AMB algorithm.

Figure 2:



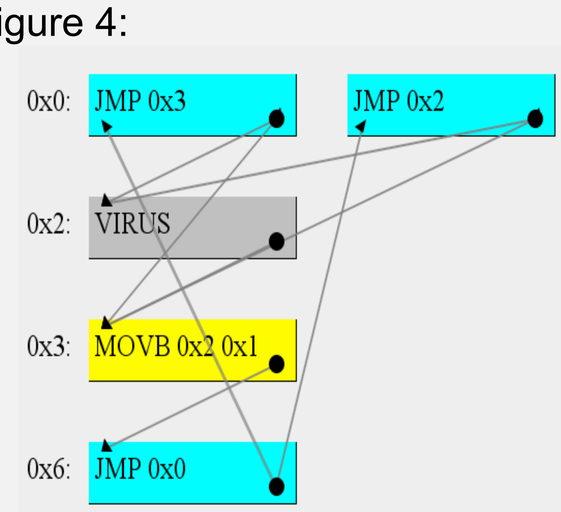
Here we demonstrate our visualization for the most basic type of linear program: a short sequence of register increments.

Figure 3:



Here we see the weakness of making an overly conservative estimate. There are thousands of impossible edges as well as unreachable instructions in this graph.

Figure 4:



Here we see the strength of the algorithm. VIRUS is not an instruction, but let it stand for a section of code that first appears to be unreachable. If instruction 0x3 is executed, it appears that instruction 0x0 always jumps to 0x3. Therefore, self-modification causes the VIRUS code to be executed.

Conclusion

With Figures 2-4, we have demonstrated the capacity of the algorithm as well as the facility of our implementation. Because of Figure 3, this algorithm’s preference to making conservative estimates makes it less useful for programs that have massive amounts of self-modification. Figure 4 demonstrates that the algorithm can present clear information that is not immediately obvious.