Identifying Input-Dependent Jumps from Obfuscated Execution using Dynamic Data Flow Graphs

Software Security, Protection, and Reverse Engineering Workshop

December 4, 2018

Joonhyung Hwang and Taisook Han

Korea Advanced Institute of Science and Technology

Overview

Introduction

Our Approach

Experimental Results

Conclusion

Introduction

Obfuscation

- Semantics-preserving program transformation
- Makes analysis difficult both for humans and machines
- Useful when you cannot trust man-at-the-end
- Used by malware authors to evade detection and analysis

Understanding Obfuscated Code

Inside-Out Approach

- Directly analyzes program behavior
- Not limited to particular obfuscation schemes

Dynamic Analysis

- Uses concrete values from program execution
- Covers only executed behavior

Identifying Branch Conditions

Input-Dependent Jumps

- Jumps whose target address depends on the input
- Decision points in program execution
- Can provide branch conditions to improve the coverage

Symbolic Execution

- Generates constraints for each execution path

The Problem

It is hard to identify input-dependent jumps and branch conditions from obfuscated execution

- Expressions for the target address are too complex
- Application of symbolic execution fails or times out

Our Contribution

Simplification of Obfuscated Execution

- Computation is represented by dynamic data flow graphs
- Non-input-dependent computation is simplified to a constant

Identification of Input-Dependent Jumps

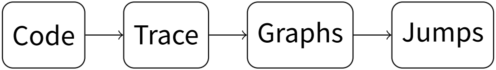
- Relation of execution before and after obfuscation is revealed
- Branch conditions are identified with reasonable effort

Overview

Our Approach

Obfuscation Mitigation

- Simplify redundant operations with constant operands
- Generate and simplify dynamic data flow graphs from traces
- Traces are generated using dynamic binary instrumentation



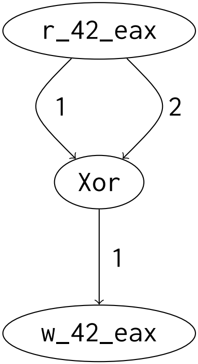
Dynamic Data Flow Graph

Directed Acyclic Multigraph

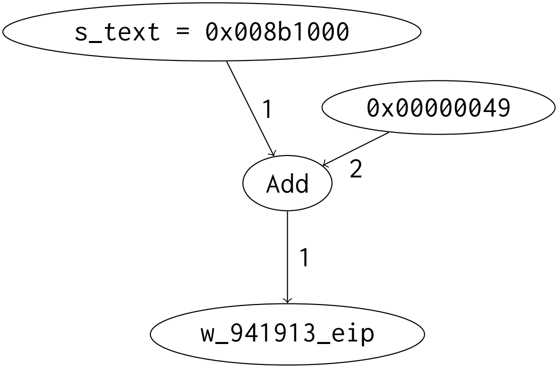
- Nodes represent computed values
- Nodes have id, type, and additional information
- Edges are directed from operands to operations
- Edges are labeled by position numbers.

Graph Examples

`42: xor eax, eax`



`941913: jmp 0x8b1049`



Graph Generation

- A graph initially has nodes for output values of interest (the target address of jumps)
- It grows by adding predecessors to nodes
- For write access, nodes are added for the operation and reading of the operands
- For read access, nodes are added for the writing of its value
  - If there is no latest writing, a node for an input variable is added
- Graphs grow until no node can be added

Graph Simplification

Simplification rules are applied until no rule can be applied

- Constant value identification
  - Value embedded in the binary
  - Value of the trap flag
- Constant value propagation
- Data movement simplification
- Operation simplification
- Nodes that do not reach an output node are removed

Simplification Rule Samples

Rules using associativity:

- $(\texttt{Add}\ x\ \ldots\ (\texttt{Add}\ y\ \ldots)) \longrightarrow (\texttt{Add}\ x\ \ldots\ y\ \ldots)$
- Same for And, Mul, Or, Xor

Like terms are combined:

- $(\texttt{Add}\ \underbrace{x\ \ldots\ x}_{20}) \longrightarrow (\texttt{Mul}\ x\ 20)$

Simplification Rule Samples

Rules using identity:

- $(\mathtt{Add}\ x\ (\mathtt{Neg}\ x)) \longrightarrow 0$, $(\mathtt{Add}\ x\ 0) \longrightarrow x$
- $(\mathtt{And}\ x\ (\mathtt{Not}\ x)) \longrightarrow 0$, $(\mathtt{And}\ x\ 0) \longrightarrow 0$, $(\mathtt{And}\ x\ x) \longrightarrow x$
- $(\mathtt{Neg}\ (\mathtt{Neg}\ x)) \longrightarrow x$, $(\mathtt{Not}\ (\mathtt{Not}\ x)) \longrightarrow x$
- $(\mathtt{Or}\ x\ 0) \longrightarrow x$, $(\mathtt{Or}\ x\ x) \longrightarrow x$
- $(\mathtt{Xor}\ x\ 0) \longrightarrow x$, $(\mathtt{Xor}\ x\ x) \longrightarrow 0$

Input-Dependent Jump Identification

A jump is input-dependent if its simplified graph has:

- a node for an outside input variable or
- a node for a result of a system-dependent operation

If an input-dependent jump is found, all access to flag operation results in the computation of the jump is considered as used

Overview

Experimental Results

- Most jumps in obfuscated execution are not input-dependent
- Numbers of identified input-dependent jumps are often same for obfuscated and original execution
- Branch condition can be understood using simplified graphs

Samples

- Factorial and bubble sort programs
  - For x86 Windows
  - Obfuscated by Code Virtualizer 1.3.9.10 and 2.2.2.0, Themida 2.4.6.0, and VMProtect 2.13.6 and 3.1.2.830
- Tigress Challenges
  - For x64 Linux

Jumps from Factorial of 10

Obfuscator	Total Jumps	Identified Jumps
Original	22	11
Code Virtualizer 1	24752	11
Code Virtualizer 2	10492	11
Themida 2	9895	887
VMProtect 2	56198	11
VMProtect 3	16785	11

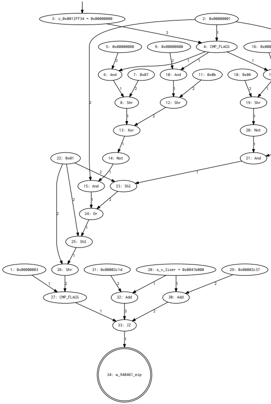
Jumps from Bubble Sort of 3, 2, and 1

Obfuscator	Total Jumps	Identified Jumps
Original	19	6
Code Virtualizer 1	33502	6
Code Virtualizer 2	12062	6
Themida 2	11350	968
VMProtect 2	35213	40
VMProtect 3	16635	6

Jumps from Tigress Challenges

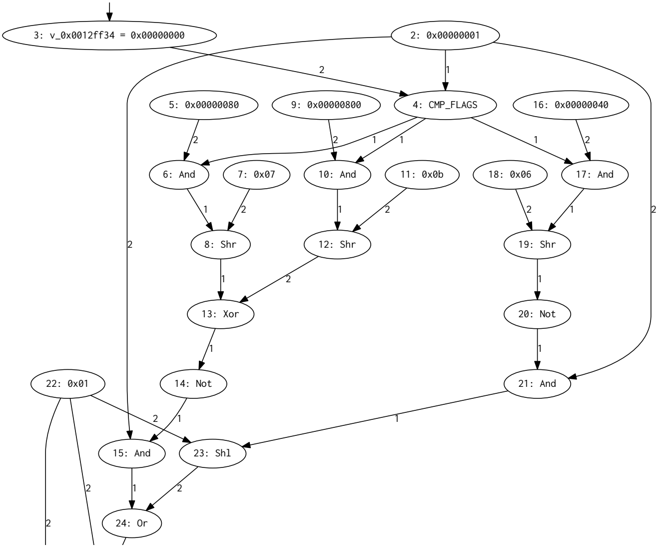
Obfuscator	Total Jumps	Identified Jumps
0000/challenge-0	2872	0
0000/challenge-1	11426	1
0000/challenge-2	10409	3
0000/challenge-3	3421	0
0000/challenge-4	2725	1
0003/challenge-0	24623	2
0003/challenge-3	3579	1

Simplified JNLE Obfuscated by Code Virtualizer 1

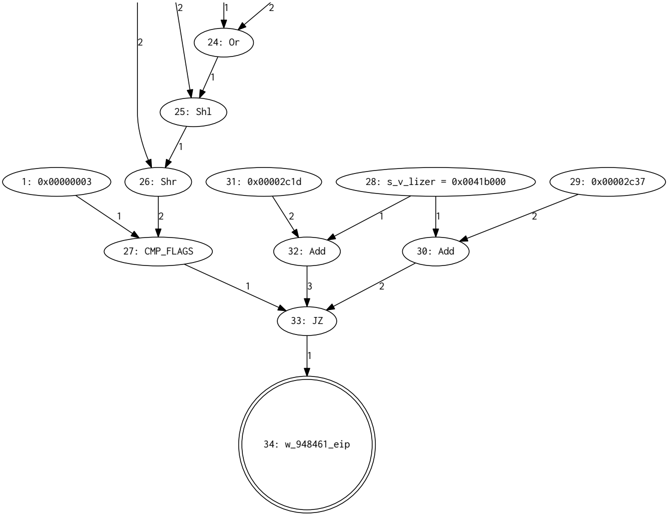


15,863 nodes and 19,717 edges $\rightarrow$ 34 nodes and 40 edges

Simplified JNLE Obfuscated by Code Virtualizer 1



Simplified JNLE Obfuscated by Code Virtualizer 1



Overview

Introduction

Our Approach

Experimental Results

Conclusion

Conclusion

- Generation and simplification of dynamic data flow graphs can remove the effect of obfuscation
- Input-dependent jumps can be used to reveal the relation between obfuscated and original execution
- Performance can be improved by using better algorithms with parallel execution
- Our work can be applied to improve other techniques such as symbolic execution
- We plan to perform further control flow analysis