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

Introduction

Our Approach

Experimental Results

Conclusion
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`

```
42: xor eax, eax
```

```
s_text = 0x008b1000
```

```
941913: jmp 0x8b1049
```

```
0x00000049
```

Our Approach

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

```
Our Approach
```

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

```
Our Approach
```

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

```
Our Approach
```

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

```
Our Approach
```
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:

- \((\text{Add } x \ldots (\text{Add } y \ldots)) \rightarrow (\text{Add } x \ldots y \ldots)\)
- Same for And, Mul, Or, Xor

Like terms are combined:

- \((\text{Add } x \ldots x) \rightarrow (\text{Mul } x \ 20)\)
Simplification Rule Samples

Rules using identity:

- \((\text{Add } x (\text{Neg } x)) \rightarrow 0, (\text{Add } x 0) \rightarrow x\)
- \((\text{And } x (\text{Not } x)) \rightarrow 0, (\text{And } x 0) \rightarrow 0, (\text{And } x x) \rightarrow x\)
- \((\text{Neg } (\text{Neg } x)) \rightarrow x, (\text{Not } (\text{Not } x)) \rightarrow x\)
- \((\text{Or } x 0) \rightarrow x, (\text{Or } x x) \rightarrow x\)
- \((\text{Xor } x 0) \rightarrow x, (\text{Xor } x x) \rightarrow 0\)
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

Introduction

Our Approach

Experimental Results

Conclusion
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

<table>
<thead>
<tr>
<th>Obfuscator</th>
<th>Total Jumps</th>
<th>Identified Jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>22</td>
<td>11</td>
</tr>
<tr>
<td>Code Virtualizer 1</td>
<td>24752</td>
<td>11</td>
</tr>
<tr>
<td>Code Virtualizer 2</td>
<td>10492</td>
<td>11</td>
</tr>
<tr>
<td>Themida 2</td>
<td>9895</td>
<td>887</td>
</tr>
<tr>
<td>VMProtect 2</td>
<td>56198</td>
<td>11</td>
</tr>
<tr>
<td>VMProtect 3</td>
<td>16785</td>
<td>11</td>
</tr>
</tbody>
</table>
### Jumps from Bubble Sort of 3, 2, and 1

<table>
<thead>
<tr>
<th>Obfuscator</th>
<th>Total Jumps</th>
<th>Identified Jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>Code Virtualizer 1</td>
<td>33502</td>
<td>6</td>
</tr>
<tr>
<td>Code Virtualizer 2</td>
<td>12062</td>
<td>6</td>
</tr>
<tr>
<td>Themida 2</td>
<td>11350</td>
<td>968</td>
</tr>
<tr>
<td>VMProtect 2</td>
<td>35213</td>
<td>40</td>
</tr>
<tr>
<td>VMProtect 3</td>
<td>16635</td>
<td>6</td>
</tr>
</tbody>
</table>
### Jumps from Tigress Challenges

<table>
<thead>
<tr>
<th>Obfuscator</th>
<th>Total Jumps</th>
<th>Identified Jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000/challenge-0</td>
<td>2872</td>
<td>0</td>
</tr>
<tr>
<td>0000/challenge-1</td>
<td>11426</td>
<td>1</td>
</tr>
<tr>
<td>0000/challenge-2</td>
<td>10409</td>
<td>3</td>
</tr>
<tr>
<td>0000/challenge-3</td>
<td>3421</td>
<td>0</td>
</tr>
<tr>
<td>0000/challenge-4</td>
<td>2725</td>
<td>1</td>
</tr>
<tr>
<td>0003/challenge-0</td>
<td>24623</td>
<td>2</td>
</tr>
<tr>
<td>0003/challenge-3</td>
<td>3579</td>
<td>1</td>
</tr>
</tbody>
</table>
15,863 nodes and 19,717 edges → 34 nodes and 40 edges
Simplified JNLE Obfuscated by Code Virtualizer 1

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

Experimental Results
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