Syntia: Synthesizing the Semantics of Obfuscated Code

Tim Blazytko  Moritz Contag  Cornelius Aschermann  Thorsten Holz

Ruhr-Universität Bochum

August 17, 2017
Code obfuscation

\[ \vec{I} = (i_1, \ldots, i_n) \quad \vec{O} = (o_1, \ldots, o_m) \]

- semantics-preserving transformation
- DRM systems, software protection systems, malware
Mixed Boolean-Arithmetic

\[ x + y + z \]

\[ (((x \oplus y) + ((x \land y) \ll 1)) \lor z) + (((x \oplus y) + ((x \land y) \ll 1)) \land z) \]

hard to simplify symbolically (NP-complete)
Virtual Machine-based obfuscation

obfuscated code is interpreted by virtual CPU
Related work

- Yadegari et al. use taint analysis and symbolic execution for deobfuscation (S&P 2015)

- Banescu et al. introduce code obfuscation against symbolic execution attacks (ACSAC 2016)

Contributions

- orthogonal approach to traditional techniques
- learn the code’s semantic based on its I/O behavior
- generic approach for trace simplification via program synthesis
Syntactic versus semantic complexity

\[
RAX = ((M_3 \ast M_2)^{M_4})
\]
Symbolic execution and program synthesis

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Simple</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Symbolic synthesis</td>
<td>Symbolic synthesis</td>
</tr>
<tr>
<td>Simple</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Complex</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

```plaintext
Symbolic execution and program synthesis

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Simple</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Symbolic synthesis</td>
<td>Symbolic synthesis</td>
</tr>
<tr>
<td>Simple</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Complex</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
```
Approach

Simplification of instruction traces

1. dissecting trace into trace windows
2. random sampling of each trace window
3. synthesis of trace windows
Trace dissection
Split at indirect control-flow transfers

mov rax, 0x8
add rax, rbx
jmp rdx
inc rax
ret
mov rdx, 0x1
ret

mov rax, 0x8
add rax, rbx
jmp rdx
inc rax
ret
mov rdx, 0x1
ret

Trace window 1          Trace window 2          Trace window 3
Random sampling

1. `mov rax, [rbp + 0x8]`
2. `add rax, rcx`
3. `mov [rbp + 0x8], rax`
4. `add [rbp + 0x8], rdx`

- inputs: $\vec{I} = (M_1, rcx, rdx)$
- outputs: $O_1, O_2$

<table>
<thead>
<tr>
<th></th>
<th>$M_1$</th>
<th>rcx</th>
<th>rdx</th>
<th>$O_1$</th>
<th>$O_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>7</td>
<td>10</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>16</td>
<td>31</td>
</tr>
<tr>
<td>1</td>
<td>120</td>
<td>27</td>
<td>0</td>
<td>147</td>
<td>147</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Synthesis of trace windows

<table>
<thead>
<tr>
<th>$M_1$</th>
<th>rcx</th>
<th>rdx</th>
<th>$O_1$</th>
<th>$O_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>10</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>15</td>
<td>16</td>
<td>31</td>
</tr>
<tr>
<td>120</td>
<td>27</td>
<td>0</td>
<td>147</td>
<td>147</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

We synthesize each output separately:

- $O_1 = M_1 + rcx$
- $O_2 = (M_1 + rcx) + rdx$
Program synthesis

- probabilistic optimization problem
- guided search towards more promising program candidates
- based on Monte Carlo Tree Search (MCTS)

General idea

Input: I/O samples from program \( P \)
- generate candidate program \( P' \) (based on prior knowledge)
- compare the I/O behavior of \( P' \) to \( P \)
- backpropagation
Running example

We want to synthesize

$$f(a, b) := a + b \mod 2^3$$

The set of I/O samples is

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>O</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Context-free grammar

$U \rightarrow U + U \mid U \ast U \mid a \mid b$

- non-terminal symbols: $U$
- a terminal symbol for each input: \{a, b\}
- sentences of the grammar are candidate programs: $a + b$
- intermediate programs contain non-terminal symbols: $U + U$

$U \Rightarrow U + U \Rightarrow U + b \Rightarrow a + b$
Which intermediate program is more promising?

1. derive a random program candidate from the intermediate program

2. compare I/O behavior to the original program

\[ U \ast U \Rightarrow \ldots \Rightarrow ((a + a) \ast (b \ast a)) \]
\[ \Rightarrow g(a, b) := ((a + a) \ast (b \ast a)) \mod 2^3 \]

\[ U + U \Rightarrow \ldots \Rightarrow (a + (b + b)) \]
\[ \Rightarrow h(a, b) := (a + (b + b)) \mod 2^3 \]

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

We come back to this in a few minutes.
Measuring output similarity

How close is the I/O behavior to the original program?

- output similarity is represented by a score
- score 1.0: equivalent output behavior for all samples
- arithmetic mean of different similarity metrics defines the score

We compare

- how close two values are numerically (arithmetic distance)
- in how many bits two values differ (Hamming distance)
- if two values are in the same range (leading/trailing zeros/ones)
Example: Hamming distance and leading zeros

\[
\text{similarity}(O, O') := \frac{\text{hamming}(O, O') + \text{lz}(O, O')}{2}
\]

<table>
<thead>
<tr>
<th>( O )</th>
<th>( O_\ast )</th>
<th>( \text{hamming} )</th>
<th>( \text{lz} )</th>
<th>( \text{similarity} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>0.67</td>
<td>0</td>
<td>0.335</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>0.34</td>
<td>0</td>
<td>0.17</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0.34</td>
<td>0</td>
<td>0.34</td>
</tr>
</tbody>
</table>

\( U \ast U: g(a, b) \)

\[
\Rightarrow \text{average similarity: 0.28}
\]

\( U + U: h(a, b) \)

\[
\Rightarrow \text{average similarity: 0.73}
\]

\( \Rightarrow \text{from } U + U \text{ derived program candidate is more promising} \)

\( \Rightarrow \text{next generated program candidate more-likely based on } U + U \text{ than } U \ast U \)
Evaluation

- simplification of Mixed Boolean-Arithmetic
  - Tigress Obfuscator
- synthesis of arithmetic VM instruction handlers
  - commercial versions of VMProtect and Themida
- ROP gadget analysis

Verification
All synthesis results have been verified by manual reverse engineering.
Mixed Boolean-Arithmetic

```c
int p10 (int v0, int v1, int v2, int v3, int v4)
{
    int r = ((~ v0) - v4);

    return r;
}
```

generated 500 random expressions

two stages of arithmetic encoding

synthesized 448 expressions (90\%) in the first run

4 seconds per synthesis task
Probabilistic synthesis behavior

![Graph showing the relationship between the number of synthesis runs and the number of synthesized expressions.](image-url)
Arithmetic VM instruction handler

```
Arithmetic VM instruction handler

mov r15, 0x200
xor r15, 0x800
mov rbx, rbp
add rbx, 0xc0
mov rbx, qword ptr [rbx]
mov r13, 1
mov rcx, 0
mov r15, ...
add r13, 0x80
add qword ptr [rdx], 0xd
jmp rbx
u64 res = M13 + M14
```
### Arithmetic VM instruction handler

<table>
<thead>
<tr>
<th></th>
<th>VMProtect</th>
<th>Themida</th>
</tr>
</thead>
<tbody>
<tr>
<td>#unique trace windows</td>
<td>449</td>
<td>106</td>
</tr>
<tr>
<td>#instructions per window</td>
<td>49</td>
<td>258</td>
</tr>
<tr>
<td>#inputs per window</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>#outputs per window</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>#synthesis tasks</td>
<td>1,123</td>
<td>1,092</td>
</tr>
<tr>
<td>I/O sampling time (s)</td>
<td>118</td>
<td>60</td>
</tr>
<tr>
<td>synthesis time per task (s)</td>
<td>3.7</td>
<td>9.1</td>
</tr>
</tbody>
</table>

- **VMProtect**: 194 out of 196 handlers (98%)
- **Themida**: 34 out of 36 handlers (no I/O samples for 2 handlers)
ROP gadget analysis

- 78 unique gadgets
- 3 inputs and 2 outputs on average
- found partial semantics for 97% of the gadgets
- synthesized 91% of the 178 outputs

inc eax
pop ebp
ret

Synthesis results:

- $O_1 = \text{eax} + 1$
- $O_2 = \text{esp} + 4$
Limitations

- trace window boundaries
- semantic complexity
- non-deterministic functions
- point functions (e.g., hash comparisons)
- confusion and diffusion (cryptography)
Conclusion

- traditional deobfuscation techniques are limited by code’s complexity
- program synthesis is limited by the code’s semantic complexity
  ⇒ succeeds where traditional approaches fail
- introduced a generic approach for trace simplification
- demonstrated that program synthesis is applicable to real-world obfuscated code
Cameron B Browne et al. ‘A Survey of Monte Carlo Tree Search Methods’. In: IEEE Transactions on Computational Intelligence and AI in Games (2012).
Monte Carlo tree search (MCTS)

Introduction

- general game playing, Computer Go
- reinforcement learning
- does not require much domain knowledge
- efficient tree search for exponential decision trees
- based on random walks and Monte Carlo simulations
- synthesis as stochastic optimization problem
Monte Carlo tree search (MCTS)

Algorithm

1. node selection
   - select best child node (exploration vs. exploitation trade-off)

2. node expansion
   - derive new game states

3. simulation
   - random playouts
   - a score represents the node’s quality

4. backpropagation
   - update the path’s quality
Monte Carlo tree search (MCTS)

Visualization

Figure: MCTS algorithm [1]
Selection

Upper confidence bound for trees (UCT)

\[ \overline{X}_j + C \sqrt{\frac{\ln n}{n_j}} \]

- average child reward: \( \overline{X}_j \)
- number of simulations (parent node): \( n \)
- number of simulations (child node): \( n_j \)
- exploration-exploitation constant: \( C \)
Selection
Simulated Annealing UCT (SA-UCT)

\[ X_j + T \sqrt{\frac{\ln n}{n_j}} \]

- dynamic parameter:  \( T = C \frac{N-i}{N} \)
- exploration-exploitation constant:  \( C \)
- maximal MCTS rounds:  \( N \)
- current MCTS round:  \( i \)

Focus shifts to exploitation over time.
Synthesis tree

```
U

U U * U U + a b

U b + U U U + + U a + U U U * +

U U * a + b a +
```
Grammar components

- addition, multiplication
- unary/binary minus
- signed/unsigned division
- signed/unsigned remainder
- logical and arithmetic shifts
- unary/binary bitwise operations
- zero/sign extend
- extract
- concat
Expression derivation

\[ U \cdot U \cdot U \quad + \quad \Leftrightarrow \quad (U + (U \cdot U)) \]

- apply random production rule to top-most-right-most \( U \)
Random playout

Algorithm

Input: Set of I/O samples $S$

1. randomly derive terminal expression $T$ from current node
2. $\text{reward} := 0$
3. for all $\vec{I}, O \in S$
   3.1 evaluate terminal expression $O' := T(\vec{I})$
   3.2 $\text{reward} := \text{similarity}(O, O') + \text{reward}$
4. return $\frac{\text{reward}}{|S|}$
## Backpropagation

### Algorithm

Input: current node $n$

1. WHILE $n \neq root$
   1.1 update the nodes average reward
   1.2 increment the nodes playout count
   1.3 $n := n.parent$