Today

- How does code obfuscation work?
- What is Monte Carlo Tree Search (MCTS)?
- How does MCTS-based program synthesis work?
- How to deobfuscate assembly code with program synthesis?
Code analysis
How do we analyse code?

- static analysis
  - disassembler
  - control-flow graphs

- dynamic analysis
  - debugging
  - instruction traces

- automated analysis

Obfuscation
Make analysis more difficult
Code obfuscation
Techniques 1/2

- disassembler/debugger traps
- packers, self-modifying code
- opaque predicates (cf. next slides)
- control-flow flattening (cf. next slides)
Code obfuscation
Techniques 2/2

- mixed Boolean-arithmetic (cf. next slides)
- data encoding
- virtual machine-based obfuscation (cf. next slides)
- white-box cryptography
Code obfuscation

Opaque predicates

(2 == 3) ? a : b

always evaluate to either true or false
Code obfuscation

Control-flow flattening

- obfuscate control-flow structure
Code obfuscation
Mixed Boolean-arithmetic

\[
\begin{align*}
x + y \\
(x \oplus y) + 2 \cdot (x \land y)
\end{align*}
\]

\[
\begin{align*}
x + y + z \\
(((x \oplus y) + ((x \land y) << 1)) \lor z) + (((x \oplus y) + ((x \land y) << 1)) \land z)
\end{align*}
\]

hard to simplify symbolically
Code obfuscation
Virtual machine-based obfuscation

- virtual CPU with custom instruction set (VM instruction handler)
- obfuscated code is interpreted by virtual CPU
Code deobfuscation
Techniques 1/2

- abstract interpretation
  - analysis in an abstract domain

- SMT-based analysis
  - detection of unsatisfiability paths

- taint analysis
  - tracking the dependencies of an input
Code deobfuscation
Techniques 2/2

- symbolic execution
  - CAS-like assembly code calculation
- program synthesis
  - learning the semantics of traces
- side-channel attacks
  - DPA, fault injection on white-box cryptography
Code deobfuscation
State-of-the-art

- works on instruction traces
- mixture of taint analysis and symbolic execution
- anti-taint analysis techniques are well known
- recent work on obfuscation attacks symbolic execution
Program synthesis for deobfuscation

Don’t care about code analysis

- previous techniques precisely analyse the underlying code
  ⇒ limited by code complexity

- program synthesis is an orthogonal approach
  - limited by the complexity of the underlying semantics
  ⇒ works for code and expressions of arbitrary complexity
Oracle-guided synthesis

Given: input/output black-box oracle

\[ I = (i_1, \ldots, i_n) \quad ? \quad O = (o_1, \ldots, o_m) \]

What happens inside?
Running example

We want to synthesise

\[ f(a, b) = a + b \mod 2^3 \]

We observe

- \( f(2, 2) = 4 \)
- \( f(4, 5) = 1 \)

The set of I/O samples is

\[ S = \{(2, 2) \rightarrow 4, (4, 5) \rightarrow 1\} \]
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 optimisation 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)

Visualisation

Figure: MCTS algorithm [2]
Selection

Upper confidence bound for trees (UCT)

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

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

\[ \bar{X}_j + T \sqrt{\frac{\ln n}{n_j}} \]

- 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.
Context-free grammar

\[ U \rightarrow U U + \mid U U \ast \mid a \mid b \]

- non-terminal symbol \( U \)
- a terminal symbol for each input
- expressions: game states (nodes)
- production rules: moves in the game
- root node \( U \)
- terminal nodes: end states of the game

\[ U \Rightarrow U U + \Rightarrow U a + \Rightarrow b a + \]
Expression derivation

\[ U \ U \ U \ * \ + \Leftrightarrow (U + (U * U)) \]

- apply random production rule to top-most-right-most \( U \)
Synthesis tree
Grammar components

- addition, multiplication
- unary/binary minus
- signed/unsigned division
- signed/unsigned remainder
- logical and arithmetic shifts
- unary/binary bitwise operations
Random playout

Algorithm

Input: Set of I/O samples $S$

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

Example: random derivations for two different nodes

\[ S = \{(2, 2) \rightarrow 4, (4, 5) \rightarrow 1\} \]

1. \[ U U * \Rightarrow U U U * * \Rightarrow U U + U U * * \Rightarrow \cdots \Rightarrow a a + b a * * \]

\[ \Rightarrow g(a, b) = ((a + a) * (b * a)) \mod (2^8) \]

\[ \Rightarrow g(2, 2) = 0 \]

2. \[ U U + \Rightarrow \cdots \Rightarrow a b b + + \]

\[ \Rightarrow h(a, b) = (a + (b + b)) \mod 2^8 \]

\[ \Rightarrow h(2, 2) = 6 \]
Similarity of outputs

Metrics

Arithmetic mean of the following metrics:

- trailing zeros
- leading zeros
- trailing ones
- leading ones
- hamming distance
- numeric distance
Similarity of outputs

Example: hamming distance and leading zeros

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

- \[ \text{similarity}(4, 0) := \frac{0.67 + 0}{2} = 0.335 \]
- \[ \text{similarity}(4, 6) := \frac{0.67 + 1.0}{2} = 0.835 \]

\[ U U + \text{ has a higher reward than } U U * \]
Backpropagation

Algorithm

Input: current node $n$

1. WHILE $n \neq root$
   1. update the nodes average reward
   2. increment the nodes playout count
   3. $n := n.parent$
Monte Carlo tree search (MCTS)

Now it should make sense

Figure: MCTS algorithm [2]
Simplification of instruction traces

Overview

Procedure

1. dissecting trace intro trace windows
2. random sampling of each trace window
3. synthesis of trace windows
Trace dissection

How to determine trace window boundaries?

- trace window boundaries impact synthesis results
  - \( x \oplus y \)
  - \((x \oplus y) + 2 \cdot (x \land y)\)
- split traces at indirect control-flow transfers
Trace dissection

Example

1 mov rax, 0x8
2 add rax, rbx
3 jmp rdx
4 inc rax
5 ret
6 mov rdx, 0x1
7 ret

Instruction trace

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

Trace window 1
Trace window 2
Trace window 3
Random sampling
Generating I/O pairs

- trace memory modifications in a window
- derive inputs and outputs
  - read-before-write principle
  - inputs: memory reads, registers
  - outputs: memory writes, registers
- generate random inputs and calculate outputs
Random sampling

Example

1. \texttt{mov rax, [rbp + 0x8]}
2. \texttt{add rax, rcx}
3. \texttt{mov [rbp + 0x8], rax}
4. \texttt{add [rbp + 0x8], rdx}

- inputs: $\vec{I} = (M_0, rcx, rdx)$
- outputs: $O_0, O_1$
  
  $O_0 = M_0 + rcx$
  
  $O_1 = (M_0 + rcx) + rdx$

- $(2, 5, 7) \rightarrow (7, 14)$
- $(1, 7, 10) \rightarrow (8, 18)$
Synthesis

\[(M_0, rcx, rdx) \rightarrow (O_0, O_1)\]

\[(2, 5, 7) \rightarrow (7, 14)\]

\[(1, 7, 10) \rightarrow (8, 18)\]

We synthesise each output separately:

\[S_{O_0} := \{(2, 5, 7) \rightarrow 7, (1, 7, 10) \rightarrow 8\}\]

\[S_{O_1} := \{(2, 5, 7) \rightarrow 14, (1, 7, 10) \rightarrow 18\}\]
Evaluation

Generic approach

- synthesis of arithmetic instruction handlers
  - VMProtect
  - Themida VMs
- simplification of mixed Boolean-arithmetic
  - Tigress Obfuscator
- ROP gadget analysis
Mixed Boolean-arithmetic

Overview

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

- generated 500 random expressions (layer 3 to 5)
- 5 input variables per expression
- two stages of arithmetic encoding (average layer 156)
- synthesised 442 expressions (88.4%) in 30 minutes
- less than 4 seconds per synthesis task
Mixed Boolean-arithmetic

DEMO
Code obfuscation
Virtual machine-based obfuscation

- virtual CPU with custom instruction set (VM instruction handler)
- obfuscated code is interpreted by virtual CPU
VMProtect
Overview

- basis of Denuvo
- stack-based VM
- performs bitwise operations with NOR gates
- 48 instructions per handler
- 2 inputs and outputs per handler
DEMO
VMProtect

Results

- 12,577 trace windows on instruction trace
- 449 of them unique
- 1123 synthesis tasks finished in less than one hour
- 3 seconds per synthesis task on average
- synthesised 19.2% of the whole trace
- synthesised 93.8% of all 184 arithmetic handlers
Themida
Overview

- register-based VM architecture
- 258 instructions per handler on average
- 10 to 15 inputs/outputs per handler
Themida

Results

- 2448 trace windows on instruction trace
- 106 unique trace windows
- Synthesis finished in 77 minutes for 1092 tasks
- 4.1 seconds per synthesis task
- Learned the semantics of 34 out of 36 (94.4%) arithmetic handlers
ROP gadget analysis

- 78 unique gadgets
- 3 inputs and 2 outputs on average
- synthesised partial semantics for 91% of the gadgets
- successful in 72% of the 178 synthesis tasks
Conclusion

- obfuscation and deobfuscation techniques
- Monte Carlo Tree Search
- MCTS-based program synthesis
- simplification of instruction traces
- evaluation on commercial obfuscators
References

David Silver et al. 'Mastering the Game of Go with Deep Neural Networks and Tree Search'. In: *Nature* (2016).

Cameron B Browne et al. 'A Survey of Monte Carlo Tree Search Methods'. In: *IEEE Transactions on Computational Intelligence and AI in Games* (2012).


References II


References III


References


References VI


