RUHR
UNIVERSITÄT
BOCHUM
RUB

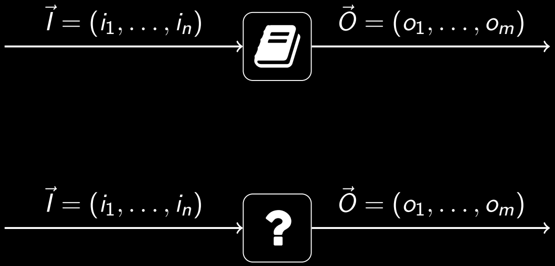
Syntia: Synthesizing the Semantics of Obfuscated Code

Tim Blazytko Moritz Contag Cornelius Aschermann Thorsten Holz

Ruhr-Universität Bochum

August 17, 2017

Code obfuscation



- semantics-preserving transformation
- DRM systems, software protection systems, malware

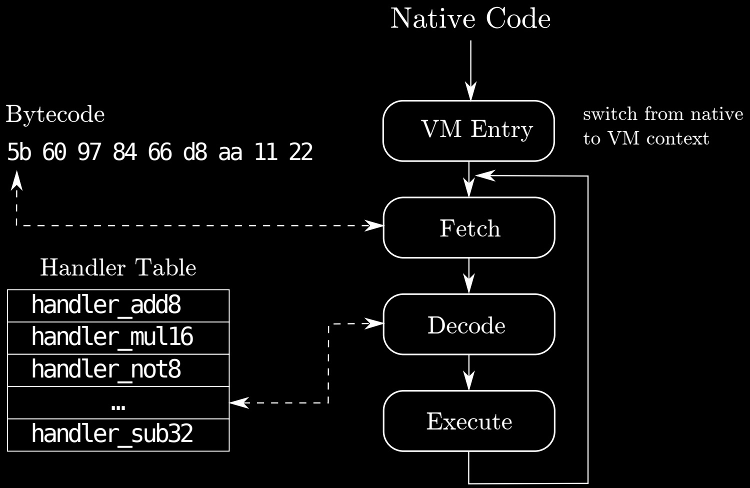
Mixed Boolean-Arithmetic

$$x + y + z$$

$$(((x \oplus y) + ((x \wedge y) << 1)) \vee z) + (((x \oplus y) + ((x \wedge y) << 1)) \wedge z)$$

hard to simplify symbolically (NP-complete)

Virtual Machine-based obfuscation



- obfuscated code is interpreted by virtual CPU

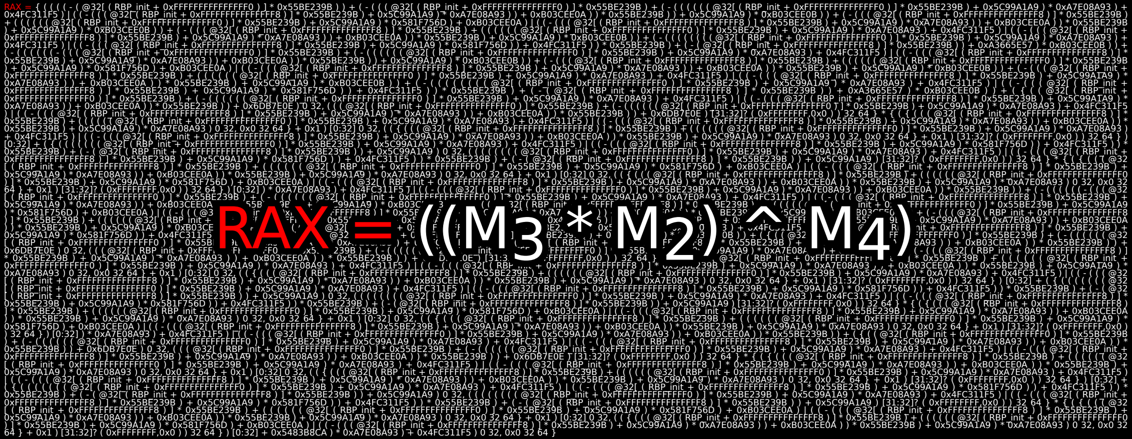
Related work

- 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



Symbolic execution and program synthesis

	semantic			
	simple		complex	
syntax	symbolic	synthesis	symbolic	synthesis
simple	✓	✓	✓	✗
complex	✗	✓	✗	✗

Approach

Simplification of instruction traces

1. dissecting trace intro 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
```

Trace window 1

```
inc rax
ret
```

Trace window 2

```
mov rdx, 0x1
ret
```

Trace window 3

Random sampling

```
mov rax, [rbp + 0x8]
add rax, rcx
mov [rbp + 0x8], rax
add [rbp + 0x8], rdx
```

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

M_1	**rcx**	**rdx**	O_1	O_2
2	5	7	7	14
1	7	10	8	18
6	10	15	16	31
120	27	0	147	147
...	...	...	...	...

Synthesis of trace windows

M_1	**rcx**	**rdx**	O_1	O_2
2	5	7	7	14
1	7	10	8	18
6	10	15	16	31
120	27	0	147	147
...	...	...	...	...

We synthesize each output separately:

- $O_1 = M_1 + \mathrm{rcx}$
- $O_2 = (M_1 + \mathrm{rcx}) + \mathrm{rdx}$

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

a	b	O
2	2	4
5	3	0
3	0	3

Context-free grammar

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

- 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: $U + U$

$$U \Rightarrow \underline{U + U} \Rightarrow U + \underline{b} \Rightarrow \underline{a} + b$$

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

- $U * U \Rightarrow \cdots \Rightarrow ((a + a) * (b * a))$

$\Rightarrow g(a, b) := ((a + a) * (b * a)) \mod 2^3$

a	b	O_*
2	2	0
5	3	6
3	0	0

- $U + U \Rightarrow \cdots \Rightarrow (a + (b + b))$

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

a	b	O_+
2	2	6
5	3	3
3	0	3

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

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

$U * U$: $g(a, b)$

O	O_*	hamming	lz	**similarity**
4	0	0.67	0	0.335
0	6	0.34	0	0.17
3	0	0.34	0	0.34

$\Rightarrow$ average similarity: 0.28

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

O	O_+	hamming	lz	**similarity**
4	6	0.67	1.0	0.835
0	3	0.34	0.34	0.34
3	3	1.0	1.0	1.0

$\Rightarrow$ average similarity: 0.73

$\Rightarrow$ from $U + U$ derived program candidate is more promising

$\Rightarrow$ next generated program candidate more-likely based on $U + U$ than $U * U$

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.

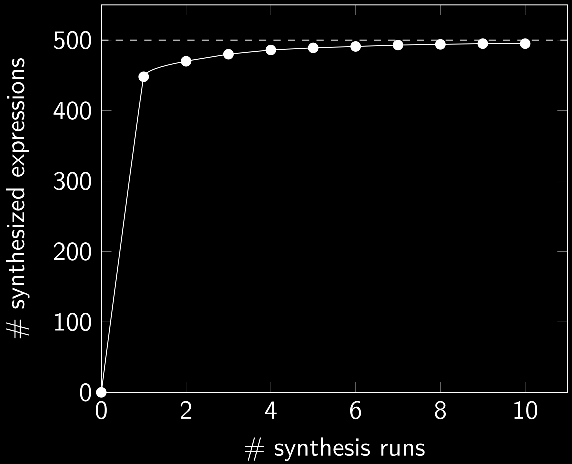
Mixed Boolean-Arithmetic

```
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



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, rbp
add r15, 0xc0
or  rcx, 0x88
add rbx, 0xb
mov r15, qword ptr [r15]
or  r12, 0xffffffff80000000
sub rcx, 0x78
movzx   r10, word ptr [rbx]
xor r12, r13
add r12, 0xffff
add r15, 0
mov r8, rbp
sub rcx, 0x10
or  r12, r12
or  rcx, 0x800
movzx   r11, word ptr [r15]
xor rcx, 0x800
mov r12, r15
add r8, 0
xor r12, 0xf0
mov rbx, 0x58
add r11, rbp
xor rbx, 0x800
and r12, 0x20
add rbx, 0x800
mov r11, qword ptr [r11]
add rbx, 1
and r12, r9
mov rdx, 1
xor r10d, dword ptr [r8]
sub r9, r11
pushfq
xor rbx, 0xf0
xor rbx, 0x800
and rdx, r8
mov r12, rbp
xor rdx, 0x20
sub rbx, 4
add r11, 0x2549b044
or  rbx, 0x78
and rdx, r10
mov rax, 0
add r12, 0x42
```

```
mov r15, rdx
xor r10d, dword ptr [r12]
sub r15, 0x800
or  rdx, 0x400
mov rsi, 0x200
mov r14, rbp
sub rsi, rsi
mov rdi, rbp
mov r8, 0x400
sub rsi, r9
sub r8, rsi
add r14, 0
add rsi, rax
and r8, 0x88
xor rsi, r14
mov rsi, rbp
add rdi, 0xc0
sub r8, rdi
add r8, 0x78
add rsi, 4
mov rcx, 0x200
mov rdi,
add dw
xor rcx,
add rcx
add rdi,
mov r8,
mov ax, word ptr [rdi]
mov r8, 1
mov rsi, rbp
and rcx, 8
sub rcx, 1
mov rcx, rdi
add rsi, 0x29
or  rcx, 8
mov r8, rsi
add rcx, 4
mov r13b, byte ptr [rsi]
cmp r13b, 0xd2
jbe 0x4f2c1e
and r8, r13
or  rcx, r13
or  rcx, 4
mov rbx, rbp
or  rcx, 4
sub rcx, 0x400
add rax, rbp
or  rcx, 0x80
add rcx, 0x80
add rbx, 0x5a
```

```
add r8, 1
or  r8, 0x78
add word ptr [rbx], r10w
mov r15, rax
sub r15, rax
pop r9
mov rcx, rbp
add rcx, 0xc0
mov rcx, qword ptr [rcx]
add rcx, 8
movzx   r10, word ptr [rcx]
mov r9, rbp
add r9, 0
xor r10d, dword ptr [r9]
and rdi, 0xffffffff80000000
sub r13, 0xf0
mov rsi, 0
sub r13, 0x20
mov rbx, rbp
or  r13, 0x88
and rcx, 8
mov si, word ptr [rbx]
or  r9, 0xffff
sub r9, 1
mov r9, rbp
mov r12, 0x58
add r9, 0
sub r13, 0x80
mov r15, r13
or  rcx, r12
xor esi, dword ptr [r9]
mov r10, rbp
add r10, 0xcc
sub r15, 0x20
xor esi, dword ptr [r10]
xor r13, 0x90
add rdi, 0x10
mov r14, rsi
mov rdx, rbp
add rdx, 0
add dword ptr [rdx], esi
xor r12, 1
mov r13, r15
```

```
or  r14, r14
mov rax, rbp
and rcx, r13
add rax, 4
sub r8, -0x80000000
add r13, 0xffff
and rcx, 0x20
mov r10, rbp
add r13, r15
add r14, r8
add r10, 0x89
xor word ptr [r10], si
xor rdx, r11
mov rsi, rbp
sub rdx, rbx
and rax, 0x40
or  rbx, 0xf0
add rsi, 0x5a
mov r8, rcx
movzx   rsi, word ptr [rsi]
add rdx, 0
movzx   r14, word ptr [r14]
mov rcx, 0x58
add rsi, rbp
xor rax, rdx
add r8, 0x80
mov r15, rsi
add r14, rbp
add r8, r15
mov rbx, 0
and rdx, 0x10
mov r14, qword ptr [r14]
add qword ptr [rsi], r14
pushfq
xor r11, r14
add r15, r14
mov r13, 0x12
mov r8, 0
and r14, 0x88
and r13, 0x40
add r13, 1
mov rdx, rbp
```

```
mov r14, 0x200
add rdx, 0xc0
add r11, r14
or  r15, 0x88
mov rdx, qword ptr [rdx]
add rdx, 0xa
add r11, 0x78
mov r8b, byte ptr [rdx]
cmp r8b, 0
je  0x4f2ede
mov rdx, rbp
or  r11, 0x40
and r15, 1
xor r11, 0x10
add rdx, 0xc0
or  r14, 4
mov r15, 0x12
mov rdx, qword ptr [rdx]
sub r11, r8
add rdx, 4
xor rsi, 0x8
xor rbx, 0xffffffff80000000
add rsi, 0x78
mov r10b, 0x68
mov r9, 0x12
or  rbx, r10
and r15, 0x78
mov r14, rbp
or  r9, 8
add r14, 0x29
xor rbx, rdi
and r15, 0x3f
or  byte ptr [r14], r10b
mov rax, 0x58
mov r8, rbp
sub rsi, 0x78
add r8, 0x127
mov rdi, rbx
xor rbx, 0x3f
mov r8, qword ptr [r8]
xor rsi, 1
mov rax, rbp
```

```
add r15, 0x3f
or  r15, 0xffffffff80000000
and rsi, r9
add rax, 0xc0
add rdi, r14
or  rsi, 1
mov rax, qword ptr [rax]
and rdi, 0x7fffffff
add rax, 2
sub rsi, 4
or  rbx, rsi
movzx   rax, word ptr [rax]
mov r9, rbp
mov r13, 0x200
mov r10, 0x58
add r9, 0
or  r10, 0x20
add eax, dword ptr [r9]
xor r10, 0x40
add eax, 0x3f505c07
add r15, 0x88
mov r12, rbp
or  rdi, 0x90
add r12, 0
or  rbx, 0x80
add rdi, 0xf0
mov r13, 0x400
add dword ptr [r12], eax
and rsi, r8
or  r10, 8
and rbx, 0x20
and rax, 0xffff
mov r11, 0
add r13, r8
or  rbx, 1
shl rax, 3
add r8, rax
or  rbx, r15
sub r15, 0x10
or  r11, r13
mov rbx, qword ptr [r8]
mov rdx, rbp
sub r13, 0x80
add rdx, 0xc0
add qword ptr [rdx], 0xd
jmp rbx
```

$$u64\ res = M_{13} + M_{14}$$

Arithmetic VM instruction handler

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

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

ROP gadget analysis

```
inc eax
pop ebp
ret
```

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

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

  $\Rightarrow$ succeeds where traditional approaches fail

- introduced a generic approach for trace simplification
- demonstrated that program synthesis is applicable to real-world obfuscated code

References I

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

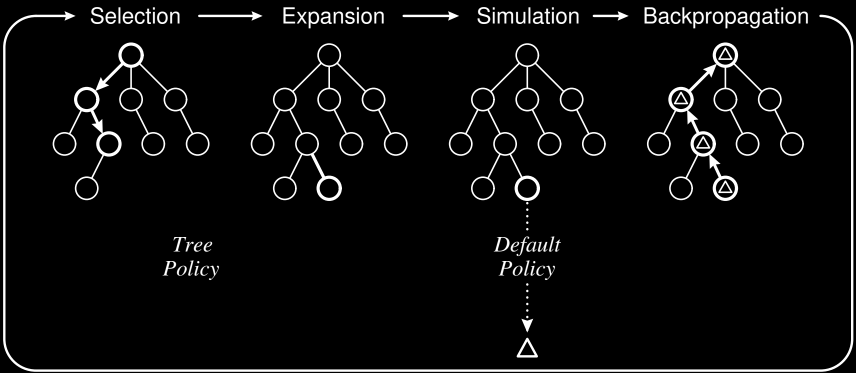


Figure: MCTS algorithm [1]

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

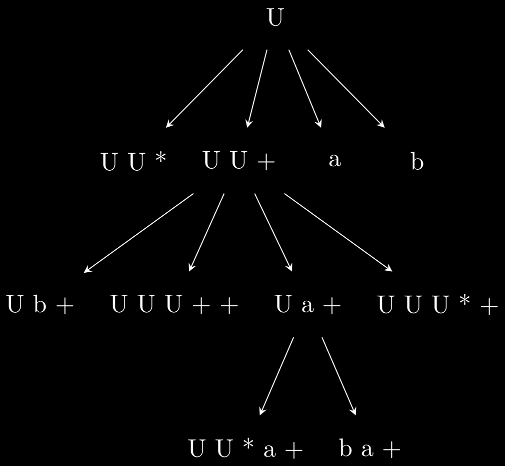
Simulated Annealing UCT (SA-UCT)

$$\overline{X}_j + T\sqrt{\frac{\ln n}{n_j}}$$

- 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

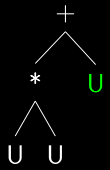


Grammar components

- 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

$$U\ U\ U\ *\ +\ \Leftrightarrow (U + (U * U))$$



- 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. $reward := 0$
3. for all $\vec{I}, O \in S$
   - 3.1 evaluate terminal expression $O' := T(\vec{I})$
   - 3.2 $reward := \text{similarity}(O, O') + reward$
4. return $\frac{reward}{|S|}$

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 $n := n.parent$