The Case for Semantics-Based Methods in Reverse Engineering

© Rolf Rolles, Funemployed
RECON 2012 Keynote
The Point of This Keynote

- Demonstrate the utility of academic program analysis towards solving real-world reverse engineering problems
Definitions

- **Syntactic methods** consider only the encoding rather than the meaning of a given object, e.g., sequences of machine-code bytes or assembly language instructions, perhaps with wildcards.

- **Semantic methods** consider the meaning of the object, e.g., the effects of one or more instructions.
Syntax vs. Semantics

• Syntactic methods
  • tend to be fast, but are limited in power
  • work well in some cases, and poorly in others
  • are incapable of solving certain types of problems

• Semantic methods
  • tend to be slower, but are more powerful
  • some analyses might produce approximate information (i.e. “maybe” instead of “yes” or “no”)

Syntax-Based Methods

- Are employed in cases such as
  - Packer entrypoint signatures
  - FLIRT signatures
  - Methods to locate functionality e.g. FindCrypt
  - Anti-virus byte-level signatures
  - Deobfuscation of pattern-obfuscated code
Syntactic Methods: Strengths

• Syntactic methods work well when the **essential feature** of the object lives in a restricted syntactic universe
  
  • FLIRT signatures in the case where the library is actually statically-distributed and not recompiled
  
  • Packer EP signatures when the packer always generates the same entrypoint
  
  • There is only one instance of some malicious software
  
  • Obfuscators with a limited vocabulary
FLIRT Signatures: Good Scenario

- Library statically-linked, not recompiled

```
6A 58          push  58h  
68 70 E4 40 00 push offset unk_40E470
E8 9A 04 00 00 call  __SEH_prolog4
33 DB          xor   ebx, ebx
89 5D E4       mov   [ebp+var_1C], ebx
89 5D FC       mov   [ebp+ms_exc.disabled], ebx
8D 45 98       lea   eax, [ebp+StartupInfo]
50             push  eax
FF 15 C0 B0 40 00 call  ds:GetStartupInfoA

6A 58          push  58h  
68 60 0A 55 00 push offset unk_550A60
E8 BB 05 00 00 call  __SEH_prolog4
33 DB          xor   ebx, ebx
89 5D E4       mov   [ebp+var_1C], ebx
89 5D FC       mov   [ebp+ms_exc.disabled], ebx
8D 45 98       lea   eax, [ebp+StartupInfo]
50             push  eax
FF 15 6C 11 51 00 call  ds:GetStartupInfoA
```
Syntactic Methods: Weaknesses

- They do not work well when there are a **variety of ways to encode the same property**
  - FLIRT signatures when the library is recompiled
  - Packer EP signatures when the packer generates the EP polymorphically
  - AV signatures for polymorphic malware, or malware distributed in source form
  - Complex obfuscators
- Making many signatures to account for the variation is not a good solution either
FLIRT Signatures: Bad Scenario

- Library was recompiled
Semantics-Based Methods

- Numerous applications in RE, including:
  - Automated key generator generation
  - Semi-generic deobfuscation
  - Automated bug discovery
  - Switch-as-binary-search case recovery
  - Stack tracking
- This keynote attacks these problems via abstract interpretation and theorem proving

```c
; and dword ptr ss:[esp], eax
T38d = load(mem37,ESP,TypeReg_32)
T39d = EAX
T40d = T38d&T39d
ZF = T40d==const(TypeReg_32,0x0)
PF =
cast(low,TypeReg_1,!( (T40d>>const(TypeReg_8,0x7))^((T40d>>const(TypeReg_8,0x5))^((T40d>>const(TypeReg_8,0x4))^((T40d>>const(TypeReg_8,0x3))^((T40d>>const(TypeReg_8,0x2))^((T40d>>const(TypeReg_8,0x1))|^T40d))))))
SF = (T40d&const(TypeReg_32,0x80000000))!=const(TypeReg_32,0x0)
CF = const(TypeReg_1,0x0)
```
Exposing the Semantics

The right-hand side is the Intermediate Language translation (or IR).

00 pop eax
01 and [esp], eax
04 pushf

label_00000000:
; pop eax
T41d = load(mem37, ESP, TypeReg_32)
ESP = ESP+const(TypeReg_32, 0x4)
EAX = T41d
label_00000001:
; and dword ptr ss:[esp], eax
T42d = load(mem37, ESP, TypeReg_32)
T43d = EAX
T44d = T42d&T43d
ZF = T44d==const(TypeReg_32, 0x0)
PF = cast(low, TypeReg_1, !(T44d>>const(T)
SF = (T44d&const(TypeReg_32, 0x80000000))
CF = const(TypeReg_1, 0x0)
OF = const(TypeReg_1, 0x0)
AF = const(TypeReg_1, 0x0)
mem37 = store(mem37, ESP, T44d, TypeReg_32)
label_00000004:
; pushfd
T45d = ((((cast(unsigned, TypeReg_32, CF)
ESP = ESP-const(TypeReg_32, 0x4)
mem37 = store(mem37, ESP, T45d, TypeReg_32)
Design of a Semantics Translator

1. Programming language-theoretic decisions
   - Tree-based? Three-address form?

2. Which behaviors to model?
   - Exceptions? Low-level details e.g. segmentation?

3. How to model those behaviors?
   - Sign flag: \((\text{result} \& 0x80000000), \text{or} (\text{result} < 0)\)?
   - Carry/overflow flags: model them as bit hacks a la Bochs, or as conditionals a la Relational REIL?

4. How to ensure correctness?
   - Easier for the programmer != better results
Act I
Old-School Program Analysis
Abstract Interpretation
Abstract Interpretation: Signs

- AI is complicated, but the basic ideas are not
- Ex: determine each variable's sign at each point

<table>
<thead>
<tr>
<th>Concrete State</th>
<th>Concrete Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 1$;</td>
<td>$x \ y \ z \ w$</td>
</tr>
<tr>
<td>$y = -1$;</td>
<td>$1, -1, ?, ?$</td>
</tr>
<tr>
<td>$z = x \times y$;</td>
<td>$1, -1, -1, ?$</td>
</tr>
<tr>
<td>$w = x \times y$;</td>
<td>$1, -1, -1, 0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstract State</th>
<th>Abstract Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = +$;</td>
<td>$x \ y \ z \ w$</td>
</tr>
<tr>
<td>$y = -$;</td>
<td>$+, -, ?, ?$</td>
</tr>
<tr>
<td>$z = x \times y$;</td>
<td>$+, -, -, ?$</td>
</tr>
<tr>
<td>$w = x \times y$;</td>
<td>$+, -, -, T$</td>
</tr>
</tbody>
</table>

- Replaced the
  - concrete state with an abstract state
  - concrete semantics with an abstract semantics
Concept: Abstract the State

- Different abstract interpretations use different abstract states.
- For the signs analysis, each variable could be:
  - Unknown: either positive or negative (+/-)
  - Positive: $x \geq 0$ (0+)
  - Negative: $x \leq 0$ (0-)
  - Zero (0)
  - Uninitialized (?)
- Ignore all other information, e.g., the actual values of variables.
Abstract multiplication follows the well-known “rule of signs” from grade school:

- A positive times a positive is positive
- A negative times a negative is positive
- A negative times a positive is negative

Note: these remarks refer to mathematical integers; machine integers are subject to overflow.
Concept: Abstract the Semantics (+)

- Positive + positive = positive.
- Negative + negative = negative.
- Negative + positive = unknown:
  - -5 + 5. Concretely, the result is 0.
  - -6 + 5. Concretely, the result is -1.
  - -5 + 6. Concretely, the result is 1.
Example: Sparse Switch Table Recovery

- Use abstract interpretation to infer case labels for switches compiled via binary search.
- Abstract domain: intervals.
Switch Tables: Contiguous, Indexed

```c
switch(x)
{
    case 0: /* ... */ break;
    case 1: /* ... */ break;
    /* ... */
    case 9: /* ... */ break;
    default: /* ... */ break;
}
```

```c
switch(x)
{
    case 0: case 2: case 4: case 6:
        printf("even\n"); break;
    case 8: printf("even\n"); break;
    case 1: case 3: case 5: case 7:
        printf("odd\n"); break;
    case 9: printf("odd\n"); break;
    default: printf("other\n"); break;
}
```
Switch Tables: Sparsely-Populated

Switch cases are sparsely-distributed.
Cannot implement efficiently with a table.

One option is to replace the construct with a series of if-statements.
This works, but takes $O(N)$ time.

Instead, compilers generate decision trees that take $O(\log(N))$ time, as shown on the next slide.
Decision Trees for Sparse Switches
Assembly Language Reification

mov eax, [ebp+arg_0]
cmp eax, 11270h
jg short loc_40167B
jz short loc_40166B
cmp eax, 3C3h
jg short loc_401654
jz short loc_401644
dec eax
jz short loc_401634
sub eax, 11
jnz loc_4016BE
push offset a000000012
call ds:__imp__printf

Additional, slight complication: red instructions modify EAX throughout the decision tree.
Assembly Language Reification, Graphical
The Abstraction

- **Insight**: we care about what **range of values** leads to a terminal case
- **Data abstraction**: Intervals $[l, u]$, where $l \leq u$
- **Insight**: construct implemented via `sub`, `dec`, `cmp` instructions – all are actually subtractions – and conditional branches
- **Semantics abstraction**: Preservation of subtraction, bifurcation upon branching
Beginning with no information about arg_0, each path through the decision tree induces a constraint upon its range of possible values, with single values or simple ranges at case labels.
Example: Generic Deobfuscation

- Use abstract interpretation to remove superfluous basic blocks from control flow graphs.
- Abstract domain: three-valued bitvectors.
Anti-Tracing Control Obfuscation

- This code is an anti-tracing check. First it pushes the flags, rotates the trap flag into the zero flag position, restores the flags, and then jumps if the zero flag (i.e., the previous trap flag) is set.

- The 90mb binary contains 10k-100k of these checks.

```
mov     edx, ss
db      66h
mov     ss, dx
pushf
pop     edx
and     edx, 100h
rol     edx, 18h
ror     edx, 1Ah
pushf
and     dword ptr [esp+0], 0FFFFFFFBFh
or      [esp+0], edx
popf
jz      loc_34EC49
```
Obfuscated Control Flow Graph

Left: control flow graph with obfuscation of the type on the previous slide.
Right: the same control flow graph with the bogus jumps removed by the analysis that we are about to present.
A Semantic Pattern for This Check

- A bit in a quantity (e.g., the TF bit resulting from a pushf instruction) is declared to be a constant (e.g., zero), and then the bit is used in further manipulations of that quantity.

- Abstractly similar to constant propagation, except instead of entire quantities, we work on the bit level.
Problem: Unknown Bits

- We only know that certain bits are constant; how do we handle non-constant ones?
- What happens if we …
  - and, adc, add, cmp, dec, div, idiv, imul, inc, mul, neg, not, or, rcl, rcr, rol, ror, sar, shl, shr, sbb, setcc, sub, test, xor
- … quantities that contain unknown bits?

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>1</th>
<th></th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Abstract Domain: Three-Valued Bitvectors

- Abstract bits as having three values instead of two: 0, 1, ½ (½ = unknown: could be 0 or 1)

- Model registers as vectors of three-valued bits
- Model memory as arrays of three-valued bytes
Abstract Semantics: AND

- Standard concrete semantics for AND:
- What happens when we introduce \( \frac{1}{2} \) bits?
- \( \frac{1}{2} \) AND 0 = 0 AND \( \frac{1}{2} \) = 0 (0 AND anything = 0)
- \( \frac{1}{2} \) AND 1 = 1 AND \( \frac{1}{2} \) = …
  - If \( \frac{1}{2} = 0 \), then 0 AND 1 = 0
  - If \( \frac{1}{2} = 1 \), then 1 AND 1 = 1
  - Conflictory, therefore \( \frac{1}{2} \) AND 1 = \( \frac{1}{2} \).
- Similarly \( \frac{1}{2} \) AND \( \frac{1}{2} \) = \( \frac{1}{2} \).
- Final three-valued truth table:
Abstract Semantics: Bitwise Operators

These operators follow the same pattern as the derivation on the previous slide, and work exactly how you would expect.
Abstract Semantics: Shift Operators

Some three-valued bitvector, call it BV

BV SHR 1

BV SHL 1

BV SAR 1

Rotation operators are decomposed into shifts and ORs, so they are covered as well.
Concrete Semantics: Addition

● How addition $C = A + B$ works on a real processor.

● $A[i], B[i], C[i]$ means the bit at position $i$.

<table>
<thead>
<tr>
<th>Carry-Out</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[i]</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B[i]</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Carry-In</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C[i]</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

● At each bit position, there are $2^3 = 8$ possibilities for $A[i]$, $B[i]$, and the carry-in bit. The result is $C[i]$ and the carry-out bit.
Abstract Semantics: Addition

- Abstractly, $A[i]$, $B[i]$, and the carry-in are three-valued, so there are $3^3$ possibilities at each position.

<table>
<thead>
<tr>
<th>Carry-Out</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>$\frac{1}{2}$</th>
<th>$\frac{1}{2}$</th>
<th>$\frac{1}{2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A[i]$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\frac{1}{2}$</td>
<td>$\frac{1}{2}$</td>
<td>$\frac{1}{2}$</td>
</tr>
<tr>
<td>$B[i]$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\frac{1}{2}$</td>
<td>$\frac{1}{2}$</td>
<td>$\frac{1}{2}$</td>
</tr>
<tr>
<td>Carry-In</td>
<td>0</td>
<td>0</td>
<td>$\frac{1}{2}$</td>
<td>$\frac{1}{2}$</td>
<td>$\frac{1}{2}$</td>
<td>0</td>
</tr>
<tr>
<td>Result</td>
<td>0</td>
<td>0</td>
<td>$\frac{1}{2}$</td>
<td>$\frac{1}{2}$</td>
<td>$\frac{1}{2}$</td>
<td>$\frac{1}{2}$</td>
</tr>
</tbody>
</table>

- The derivation is straightforward but tedious.

- Notice that the system automatically determines that the sum of two $N$-bit integers is at most $N+1$ bits.
Abstract Semantics: Negation, Subtraction

- \( \text{Neg}(x) = \text{Not}(x)+1 \)
- \( \text{Sub}(x,y) = \text{Add}(x,\sim y) \) where the initial carry-in for the addition is set to one instead of zero.
- Therefore, these operators can be implemented based upon what we presented already.
Unsigned Multiplication

- Consider \( B = A \times 0x123 \)
- \( 0x123 = 0001\ 0010\ 0011 = 2^8 + 2^5 + 2^1 + 2^0 \)
- \( B = A \times (2^8 + 2^5 + 2^1 + 2^0) \) (substitution)
- \( B = A \times 2^8 + A \times 2^5 + A \times 2^1 + A \times 2^0 \) (distributivity: \( \times \) over \( + \))
- \( B = (A << 8) + (A << 5) + (A << 1) + (A << 0) \) (definition of \( << \))

- Whence unsigned multiplication reduces to previously-solved problems
- Signed multiplication is trickier, but similar
Abstract Semantics: Conditionals

- For equality, if any concrete bits mismatch, then A != B is true, and A == B is false.

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>½</td>
<td>¹</td>
<td>½</td>
<td>½</td>
<td>½</td>
<td>0</td>
<td>½</td>
</tr>
<tr>
<td>B</td>
<td>½</td>
<td>0</td>
<td>½</td>
<td>½</td>
<td>½</td>
<td>0</td>
<td>½</td>
</tr>
</tbody>
</table>

- For A < B, compute B-A and take the carry-out as the result

- For A <= B, compute (A < B) | (A == B).
Deobfuscation Procedure

• Generate control flow graph

1. Apply the analysis to each basic block
2. If any conditional jump becomes unconditional, remove the false edge from the graph
3. Prune all vertices with no incoming edges (DFS)
4. Merge all vertices with a sole successor, whose successor has a sole predecessor
5. Iterate back to #1 until the graph stops changing

• Stupid algorithm, could be majorly improved
Progressive Deobfuscation

Original graph: 232 vertices

Deobfuscation round #1: five vertices

Deobfuscation round #2, final: one vertex
Example: Tracking ESP

- We explore and generalize Ilfak's work on stack tracking.
- Abstract domains: convex polyhedra and friends in the relational domain family.
Concept: Relational Abstractions

- So far, the analyses treated variables separately; we now consider analyses that treat variables in combination.

- Below: two-dimensional convex polyhedra induced by linear inequalities over $x$ and $y$.
Stack Tracking, Ilfak 2006

- Want to know the differential of ESP between function begin and every point in the function.
- Problem: indirect calls with unknown calling conventions.

```
lea   ecx, [esp+0C4h+var_A8]  ; esp_delta = x
push  ecx                     ; esp_delta = x
push  ebx                     ; esp_delta = x+4
push  ebx                     ; esp_delta = x+8
push  1012h                   ; esp_delta = x+12
push  offset off_546AD8      ; esp_delta = x+16
push  eax                     ; esp_delta = x+20
call  edx                     ; esp_delta = x+24
mov   eax, [esi+4]           ; esp_delta = ?????
```
Stack Tracking

- Generate a convex polyhedron, defined by:
  - Two variables for every block: in.esp, out.esp.
  - One equality for each initial and terminal block.
  - One equality for each edge (#i,#j): out.esp_i = in.esp_j
  - One inequality (not shown) for each block #n, relating in.esp_n to out.esp_n, based on the semantics (ESP modifications: calls, pushes, pops) of the block.

- Solve the equation system for an assignment to the ESP-related variables.
This block pushes 6 DWORDs (24 bytes) on the stack, and it is unknown whether the call removes them. Therefore, the inequality generated for this block is:

\[ \text{out}_\text{esp}_5 - \text{in}_\text{esp}_5 \leq 24 \]
Alternative Formulations

- Ilfak's solution uses polyhedra, which is potentially computationally expensive

- Note: all equations are of the form $v_i - v_j \leq c_{ij}$, which can be solved in $O(|V|*|E|)$ time with Bellman-Ford (or other PTIME solutions)

Figure stolen from Antoine Mine's Ph.D. thesis due to lack of time. Sorry.
Random Concept: Reduced Product

- Instead of performing analyses separately, allow them to interact => increased precision

- Suppose we perform several analyses, and the results for variable x at some point are:
  - $x = [-10,6]$ (Interval)
  - $x = 0+$ (Sign)
  - $x = \text{Odd}$ (Parity)

- Using the other domains, we can refine the interval abstraction:
  - Reduced product of $([-10,6],0+)$ = $([0,6],0+)$
  - Reduced product of $([0,6],\text{Odd})$ = $([1,5],\text{Odd})$
Act II
New-School Program Analysis
SMT Solving
Concept: Input Crafting via Theorem Proving

• Idea: convert portions of code into logical formulas, and use mathematically precise techniques to prove properties about them.

• Example: what value must EAX have at the beginning of this snippet in order for EAX to be 0x12345678 after the snippet executes?

```
sub bl, bl
movzx ebx, bl
add ebx, 0BBBBBBBBBh
add eax, ebx
```
Part of the IR translation of the x86 snippet given on the previous slide.

A slightly simplified (read: incorrect) SMT QF_EUFBV translation of the IR from the left.
Ask a Question

- Given the SMT formula, initial EAX unspecified, is it possible that this postcondition is true?
  - assert(T175d == 0x12345678); (T175d is final EAX)

- The SMT solver outputs a model that satisfies the constraints.
- The first red line says that the formula is satisfiable, i.e., the answer is yes.
- The final red line says that the initial value of EAX must be 1450744509, or 0x56789ABD.
Automated Key Generator Generation

As before, generate an execution trace (statically) and convert to IR. Then convert the IR to an SMT formula.

**Precondition:**
\[
a_{ActivationCode}[0] = X \land \ldots \land a_{ActivationCode}[2] = Z \ldots
\]
where
\[
X = \text{regcode}[0],
Y = \text{regcode}[1],
Z = \text{regcode}[2], \ldots
\]

**Postcondition:**
\[
\text{String\_derived}[0] = '0' \land \ldots \land \text{String\_derived}[2] = 'o' \ldots
\]
Example: Equivalence Checking for Error Discovery

- We employ a theorem prover (SMT solver) towards the problem of finding situations in which virtualization obfuscators produce incorrect translations of the input.
Concept: Equivalence Checking

- **Population counting**, naively. Count the number of one-bits set.

```c
for(uint i = 1; i; i <<= 1)
count += (val & i) != 0;
```
```
c00 = val & 0x00000001 ? 1 : 0;
c01 = val & 0x00000002 ? c00+1 : c00;
/* ... */
c31 = val & 0x80000000 ? c30+1 : c30;
```

Iterative bit-tests
Sequential ternary operator
Population Count via Bit Hacks

- Looks crazy; the next slide will demonstrate how this works

```assembly
mov eax, ebx
and eax, 55555555h
shr ebx, 1
and ebx, 55555555h
add ebx, eax
mov eax, ebx
and eax, 33333333h
shr ebx, 2
and ebx, 33333333h
add ebx, eax
mov eax, ebx
and eax, 0F0F0F0Fh
shr ebx, 4
and ebx, 0F0F0F0Fh
add ebx, eax
mov eax, ebx
and eax, 0FF00FFh
shr ebx, 8
and ebx, 0FF00FFh
add ebx, eax
mov eax, ebx
and eax, 0FFFFFh
shr ebx, 10h
and ebx, 0FFFFFh
add ebx, eax
mov eax, ebx
```
# 8-Bit Population Count via Bit Hacks

<table>
<thead>
<tr>
<th>Round #1</th>
<th>Round #2</th>
<th>Round #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c d e f g h &amp; 1 0 1 0 1 0 1 0 &gt;&gt;1 0 a 0 c 0 e 0 g</td>
<td>i i j j k k l l &amp; 1 1 0 0 1 1 0 0 &gt;&gt;2 0 0 i i 0 0 k k</td>
<td>m m m m n n n n &amp; 1 1 1 1 0 0 0 0 &gt;&gt;4 0 0 0 0 m m m m</td>
</tr>
<tr>
<td>&amp; 0 1 0 1 0 1 0 1 0 b 0 d 0 f 0 h</td>
<td>0 0 i i 0 0 k k</td>
<td>0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1</td>
</tr>
<tr>
<td>0 a 0 c 0 e 0 g + 0 b 0 d 0 f 0 h</td>
<td>0 0 j j 0 0 l l</td>
<td>0 0 0 0 n n n n + 0 0 0 0 n n n n</td>
</tr>
<tr>
<td>i i j j k k l l = i i j j k k l l</td>
<td>m m m m m n n n n = m m m m n n n n</td>
<td>p p p p p p p p p p</td>
</tr>
</tbody>
</table>

Where

\[
\begin{align*}
ii &= a+b \\
nn &= c+d \\
nn &= e+f \\
nn &= g+h
\end{align*}
\]

\[
\begin{align*}
ii &= a+b+c+d+e+f+g+h
\end{align*}
\]

This is the population count.
Equivalence of Naïve and Bit Hack

Convert left sequence to IR.
Assert that val = EBX.
Query whether c31 == final EAX.
Answer: YES; the sequences are equivalent.
Example: Equivalence Checking for Verification of Deobfuscation

- Given some deobfuscation procedure, we want to ensure that the output is equivalent to the input
Is this … (1 of 2)

lodsbyte ptr ds:[esi]
sub esp, 00000004h
mov DWORD ptr ss:[esp], ecx
mov cl, E3h
not cl
shr cl, 05h
sub cl, 33h
xor cl, ACh
sub cl, 94h
add al, D5h
add al, cl
sub al, D5h
mov ecx, DWORD ptr ss:[esp]
push ebx
mov ebx, esp
add ebx, 0000004h
add ebx, 00000004h
xchg DWORD ptr ss:[esp], ebx
pop esp
add al, bl
sub al, CDh
push cx
push ebx
mov bh, B7h
mov ch, bh

pop ebx
sub al, 19h
push ebx
push ecx
mov ch, 91h
mov b1, 2Fh
xor b1, ch
pop ecx
add b1, 52h
sub b1, FCh
add al, b1
pop ebx
sub al, ch
sub al, 14h
add al, 19h
pop cx
push edx
mov dl, 4Dh
add dl, 01h
add dl, 7Dh
push 0000040Eh
mov DWORD ptr ss:[es], ebx
mov b1, 11h
inc b1
add b1, F0h

sub dl, bl
pop ebx
neg dl
inc dl
push ecx
mov cl, 38h
or cl, ADh
add cl, B8h
add dl, cl
pop ecx
sub al, 5Ch
sub al, dl
add al, 5Ch
pop edx
push edx
mov dh, 41h
push ecx
mov cl, 71h
inc cl
not cl
shl cl, 02h
push eax
mov ah, 85h
mov ah, C9h
push ebx
mov bl, D2h
inc bl
dec bl
dec bl
and bl, 09h
or bl, 89h
sub bl, B6h
xor ah, bl
pop ebx
xor cl, ah
pop eax
sub cl, 46h
add dh, cl
pop ecx
sub dh, CEh
add bl, dh
pop edx
add bl, al
push edx
mov dh, DCh
shr dh, 02h
and dh, 3Eh
or dh, 3Bh
sub dh, A8h
sub bl, dh

pop edx
push 0000593Ch
mov dword ptr ss:[esp], ebx
mov ebx, 19B36B5Eh
push edx
mov edx, 57792DD8h
add ebx, edx
mov edx, dword ptr ss:[esp]
add esp, 00000004h
add ebx, 2BC3456Bh
or ebx, 6A8A718Ch
shr ebx, 03h
neg ebx
add ebx, 1FDE002Dh
add ebx, 2EC02C7Ch
add ebx, edi
sub ebx, 2EC02C7Ch
mov byte ptr ds:[ebx], al
pop ebx
... Equivalent to This?

```
lo dsb byte ptr ds:[esi]
add al, bl
sub al, B7h
sub al, ADh
add bl, al
mov byte ptr ds:[edi+00000038], al
```

Theorem prover says: **YES**, if we ignore the values below terminal ESP
Inequivalence #1

push dword ptr ss:[esp]
mov eax, dword ptr ss:[esp]
add esp, 00000004h
sub esp, 00000004h
mov dword ptr ss:[esp], ebp
mov ebp, esp
add ebp, 00000004h
add ebp, 00000004h
xchg dword ptr ss:[esp], ebp
mov esp, dword ptr ss:[esp]
inc dword ptr ss:[esp]
pushfd

Deobfuscated handler.

Obfuscated version of inc dword handler.

These sequences are INEQUIVALENT: the obfuscated version modifies the carry flag (with the add and sub instructions) before the inc takes place, and the inc instruction does not modify that flag.
Inequivalence #2

The sar instruction does not change the flags if the shiftnand is zero, whereas the obfuscated handler does change the flags via the add instructions.

Obfuscated version of sar dword handler.

```
mov cx, word ptr ss:[esp]
push edx
push esp
pop edx
push ebp
mov ebp, 00000004h
add edx, ebp
pop ebp
add edx, 00000002h
xchg dword ptr ss:[esp], edx
mov esp, dword ptr ss:[esp]
sar dword ptr ss:[esp], cl
pushfd
```

Deobfuscated handler.

```
pop cx
sar dword ptr ss:[esp], cl
pushfd
```

The sar instruction does not change the flags if the shiftnand is zero, whereas the obfuscated handler does change the flags via the add instructions.
Inequivalence #3

```
lodsd dword ptr ds:[esi]
sub eax, 773B7B89h
sub eax, ebx
add eax, 33BE2518h
xor ebx, eax
push dword ptr ds:[eax]
```

Can't show obfuscated version due to it being 82 instructions long. Obfuscated version writes to stack whereas deobfuscated version does not; therefore, the memory read on the last line could read a value below the stack pointer, which would be different in the obfuscated and deobfuscated version.
Warning: Here Be Dragons

• I tried to make my presentation friendly; the literature does not make any such attempt

**Definition 3** \( \mathcal{T}^{Ph} : \wp(\mathbb{P}) \rightarrow \wp(\mathbb{P}) \) is given by the point-wise extension of:

\[
\mathcal{T}^{Ph}(P_0) = \left\{ P_l \mid P_l = (m_l, a_l), \sigma = \sigma_0 \ldots \sigma_{l-1} \sigma_l \in S[P_0], \sigma_l = \langle a_l, m_l, \theta_l, \mathcal{J}_l \rangle, \right. \\
(\sigma_{l-1}, \sigma_l) \in MT(P_0), \forall i \in [0, l-1[: (\sigma_i, \sigma_{i+1}) \notin MT(P_0) \left\} \right.
\]

\( \mathcal{T}^{Ph} \) can be extended to traces \( \mathcal{F}_{\mathcal{T}^{Ph}}[P_0] : \wp(\mathbb{P}^*) \rightarrow \wp(\mathbb{P}^*) \) as: \( \mathcal{F}_{\mathcal{T}^{Ph}}[P_0](Z) = P_0 \cup \{ zP_i P_j \mid P_j \in \mathcal{T}^{Ph}(P_i), zP_i \in Z \} \).

**Theorem 1** \( lfp \subseteq \mathcal{F}_{\mathcal{T}^{Ph}}[P_0] = S^{Ph}[P_0] \).

A program \( Q \) is a metamorphic variant of a program \( P_0 \), denoted \( P_0 \sim_{Ph} Q \), if \( Q \) is an element of at least one sequence in \( S^{Ph}[P_0] \).

**Correctness and completeness of phase semantics.** We prove the correctness of phase semantics by showing that it is a sound approximation of trace semantics, namely by providing a pair of adjoint maps \( \alpha_{Ph} : \wp(\Sigma^*) \rightarrow \wp(\mathbb{P}^*) \) and \( \gamma_{Ph} : \wp(\mathbb{P}^*) \rightarrow \wp(\Sigma^*) \), for which the fixpoint computation of \( \mathcal{F}_{\mathcal{T}^{Ph}}[P_0] \) approximates the fixpoint computation of \( \mathcal{F}_{\mathcal{T}}[P_0] \). Given \( \sigma = \langle a_0, m_0, \theta_0, \mathcal{J}_0 \rangle \ldots \sigma_{i-1} \sigma_i \ldots \sigma_n \) we define \( \alpha_{Ph} \) as:
References

• A program analysis reading list that I compiled
  - http://www.reddit.com/r/ReverseEngineering/comments/smf4u/reverser_wanting_to_develop_mathematically/c4fa6yl
• Rolles: Switch as Binary Search
  - https://www.openrce.org/blog/view/1319/
  - https://www.openrce.org/blog/view/1320/
• Rolles: Control Flow Deobfuscation via Abstract Interpretation
  - https://www.openrce.org/blog/view/1672/
• Rolles: Finding Bugs in VMs with a Theorem Prover
• Rolles: Semi-Automated Input Crafting
  - https://www.openrce.org/blog/view/2049/
• Ilfak: Simplex Method in IDA Pro
  - http://www.hexblog.com/?p=42
Questions?

- Hopefully pertinent ones
- rolf.rolles at gmail
Thanks

- Jamie Gamble, Sean Heelan, Julien Vanegue, William Whistler
- All reverse engineers who publish
  - Especially on the RE reddit
- RECON organizers