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

```
8B 53 40        mov     edx, [ebx+40h]
8D 0C B6        lea     ecx, [esi+esi*4]
8D 0C 4E        lea     ecx, [esi+ecx*2]
8B 12           mov     edx, [edx]
89 44 8A 24     mov     [edx+ecx*4+24h], eax
46              inc     esi
8B 45 94        mov     eax, [ebp+var_6C]
3B F0           cmp     esi, eax
7C C3           jl      short loc_1301CCC7
```

- 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

```
55                   push   ebp
8B EC                mov    ebp, esp
51                   push   ecx
8B 45 08             mov    eax, [ebp+arg_0]
89 45 FC             mov    [ebp+var_4], eax
83 7D FC 09          cmp    [ebp+var_4], 9
0F 87 B0 00 00 00    ja     loc_4010C4
8B 4D FC             mov    ecx, [ebp+var_4]
FF 24 8D D8 10 40+   jmp    ds:off_4010D8[ecx*4]
```

```
55                   push   ebp
8B EC                mov    ebp, esp
8B 45 08             mov    eax, [ebp+arg_0]
83 F8 09             cmp    eax, 9
0F 87 A7 00 00 00    ja     loc_4010B6
FF 24 85 C8 10 40+   jmp    ds:off_4010C8[eax*4]
```

Semantics-Based Methods

```
; 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,
0x6))^((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)
```

- 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

Exposing the Semantics

```
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)
```

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

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: (result & 0x80000000), or (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 Analysis

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

Concrete

Semantics	State (x y z w)
$x = 1;$	$\langle 1, ?, ?, ? \rangle$
$y = -1;$	$\langle 1, -1, ?, ? \rangle$
$z = x * y;$	$\langle 1, -1, -1, ? \rangle$
$w = x + y;$	$\langle 1, -1, -1, 0 \rangle$

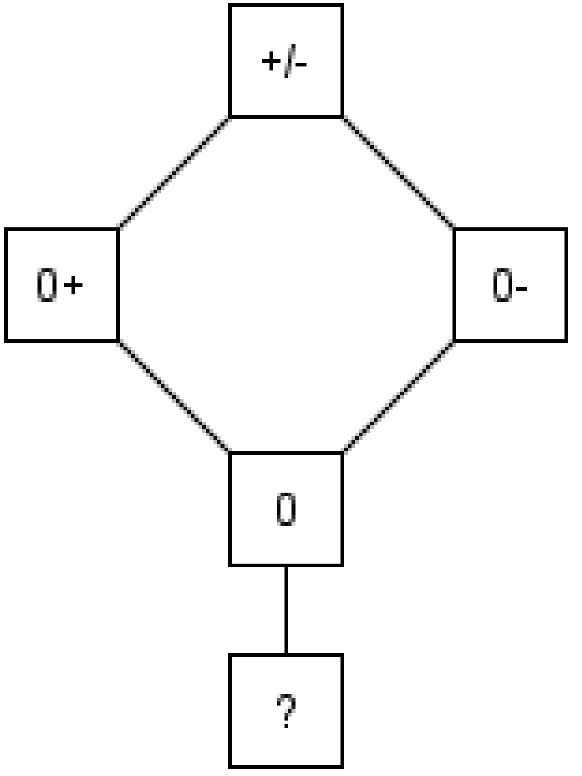
Abstract

Semantics	State (x y z w)
$x = +;$	$\langle +, ?, ?, ? \rangle$
$y = -;$	$\langle +, -, ?, ? \rangle$
$z = x *^\sharp y;$	$\langle +, -, -, ? \rangle$
$w = x +^\sharp y;$	$\langle +, -, -, \top \rangle$

- 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 >= 0 (0+)
  - Negative: x <= 0 (0-)
  - Zero (0)
  - Uninitialized (?)



- Ignore all other information, e.g., the actual values of variables.

Concept: Abstract the Semantics (*)

- 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

*	?	0	0+	0-	+/-
?	+/-	0	+/-	+/-	+/-
0	0	0	0	0	0
0+	+/-	0	0+	0-	+/-
0-	+/-	0	0-	0+	+/-
+/-	+/-	0	+/-	+/-	+/-

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.

+	?	0	0+	0-	+/-
?	+/-	+/-	+/-	+/-	+/-
0	+/-	0	0+	0-	+/-
0+	+/-	0+	0+	+/-	+/-
0-	+/-	0	0-	0+	+/-
+/-	+/-	0	+/-	+/-	+/-

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;
}
```

```
cmp     eax, 9             ; switch 10 cases
ja      loc_4010B6         ; default
jmp     ds:off_4010C8[eax*4] ; switch jump

off_4010C8 dd offset loc_401016
dd offset loc_401026
dd offset loc_401036
dd offset loc_401046
dd offset loc_401056
dd offset loc_401066
```

```c
switch(x)
{
  case 0: case 2: case 4: case 6:
  case 8:  printf("even\n"); break;

  case 1: case 3: case 5: case 7:
  case 9:  printf("odd\n");  break;

  default: printf("other\n"); break;
}
```

```
cmp     eax, 9             ; switch 10 cases
ja      short loc_401129 ; default
movzx   eax, ds:index_table[eax]
jmp     ds:off_40113C[eax*4] ; switch jump

off_40113C      dd offset loc_401109    ; DATA
                dd offset loc_401119    ; jump
index_table     db      0,      1,      0,      1
                db      0,      1,      0,      1
                db      0,      1
```

Switch Tables: Sparsely-Populated

```c
switch(x)
{
  case       1: /*1*/ break;
  case      15: /*2*/ break;
  case     973: /*3*/ break;
  case    4772: /*4*/ break;
  case   50976: /*5*/ break;
  case  661034: /*6*/ break;
  case 8109257: /*7*/ break;
}
```

```c
if(x ==       1) /*1*/ else
if(x ==      15) /*2*/ else
if(x ==     973) /*3*/ else
if(x ==    4772) /*4*/ else
if(x ==   50976) /*5*/ else
if(x ==  661034) /*6*/ else
if(x == 8109257) /*7*/;
```

Switch cases are sparsely-distributed.

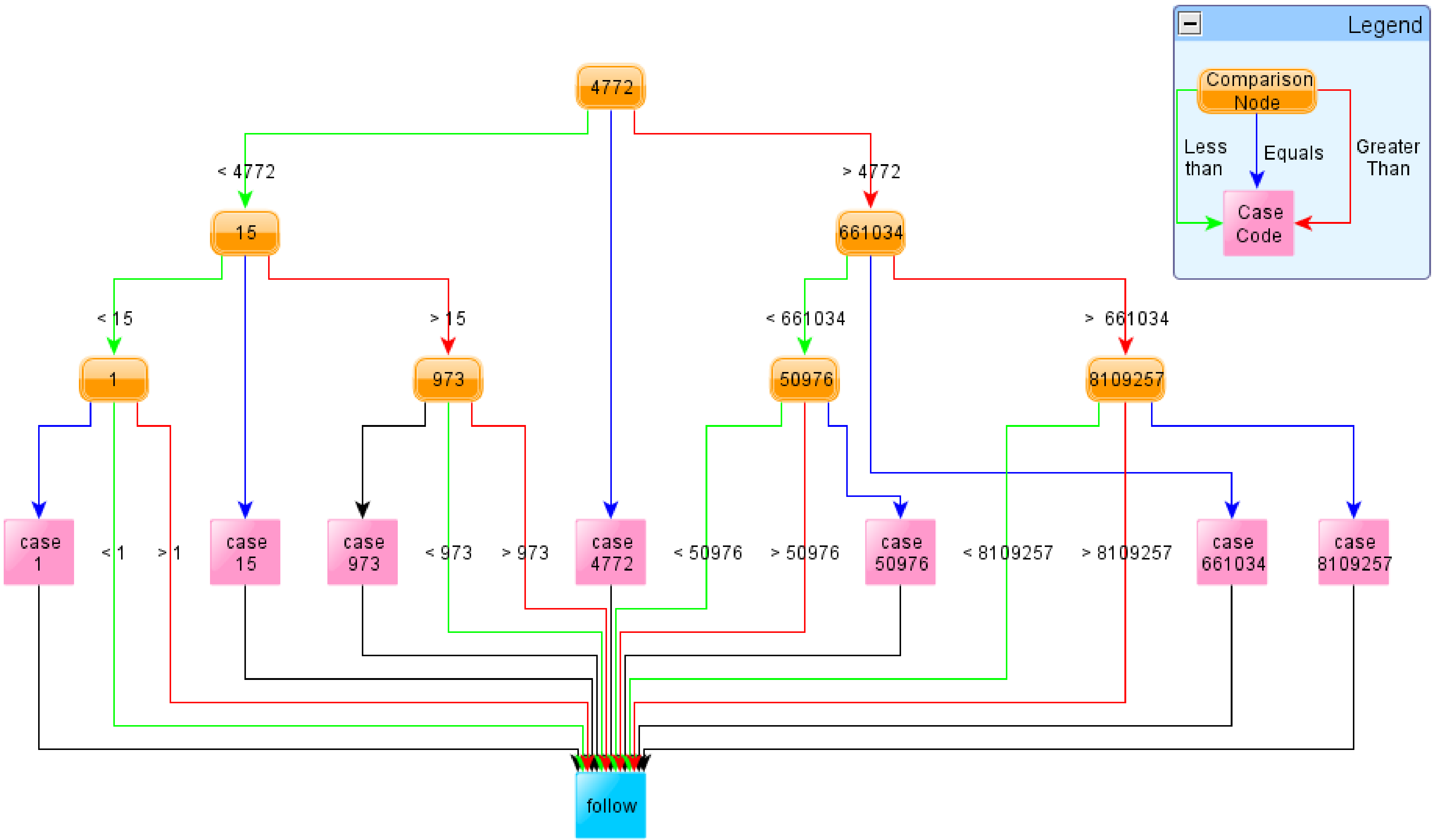
Cannot implement efficiently with a table.

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

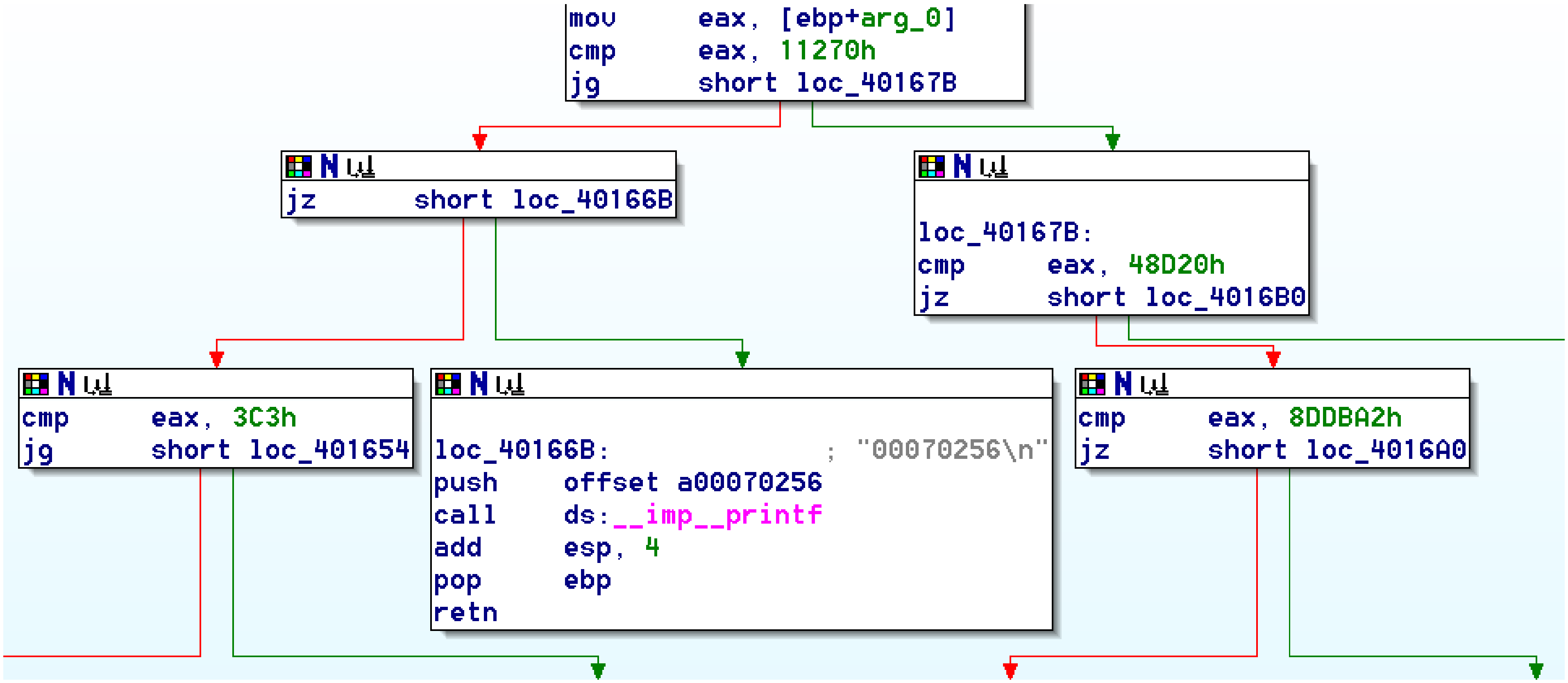


Assembly Language Reification

```
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 a00000012
call    ds:__imp__printf
```

Additional, slight complication: red instructions modify EAX throughout the decision tree.

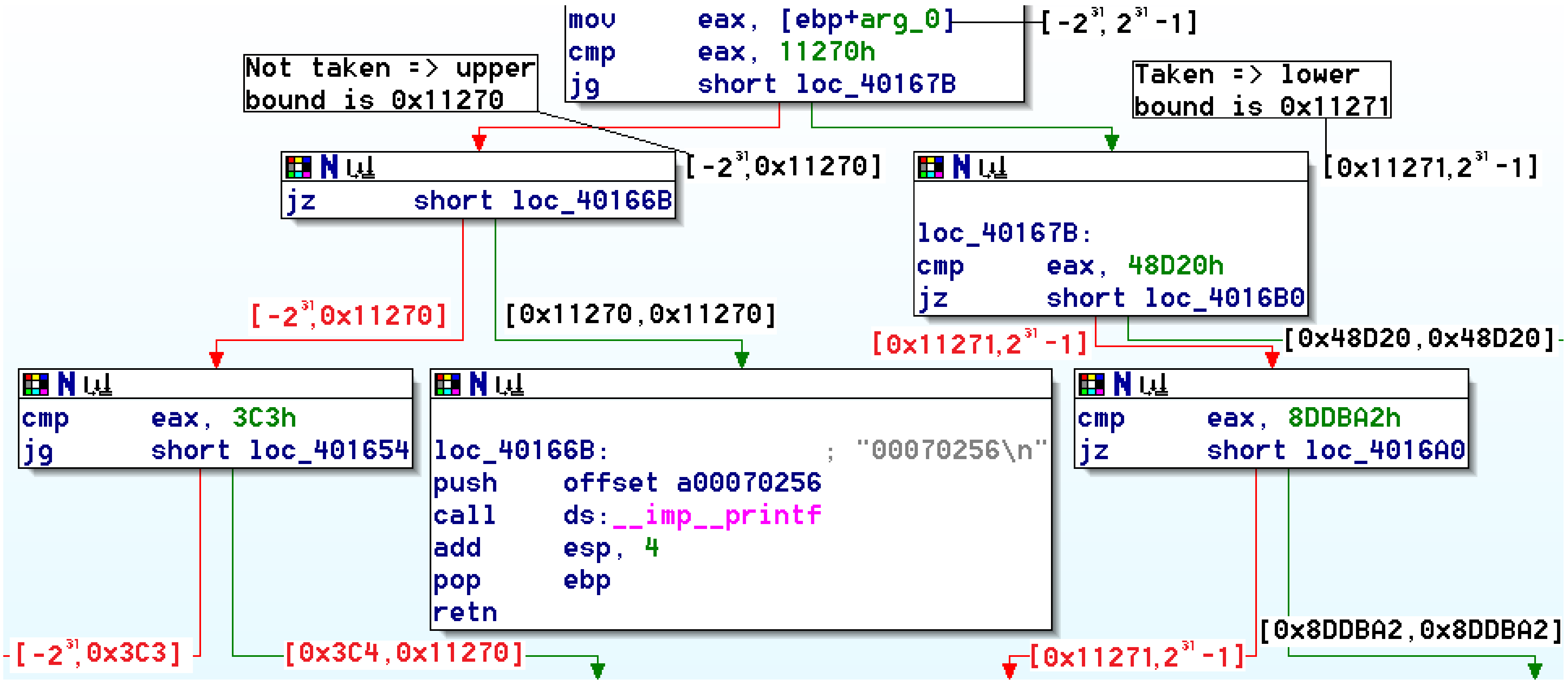
Assembly Language Reification, Graphical



The Abstraction

- *Insight*: we care about what **range of values** leads to a terminal case
- *Data abstraction*: Intervals [*l*,*u*], where *l* <= *u*
- *Insight:* construct implemented via ***sub***, ***dec***, ***cmp*** instructions – all are actually subtractions – and conditional branches
- *Semantics abstraction:* Preservation of subtraction, bifurcation upon branching

Analysis Results



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

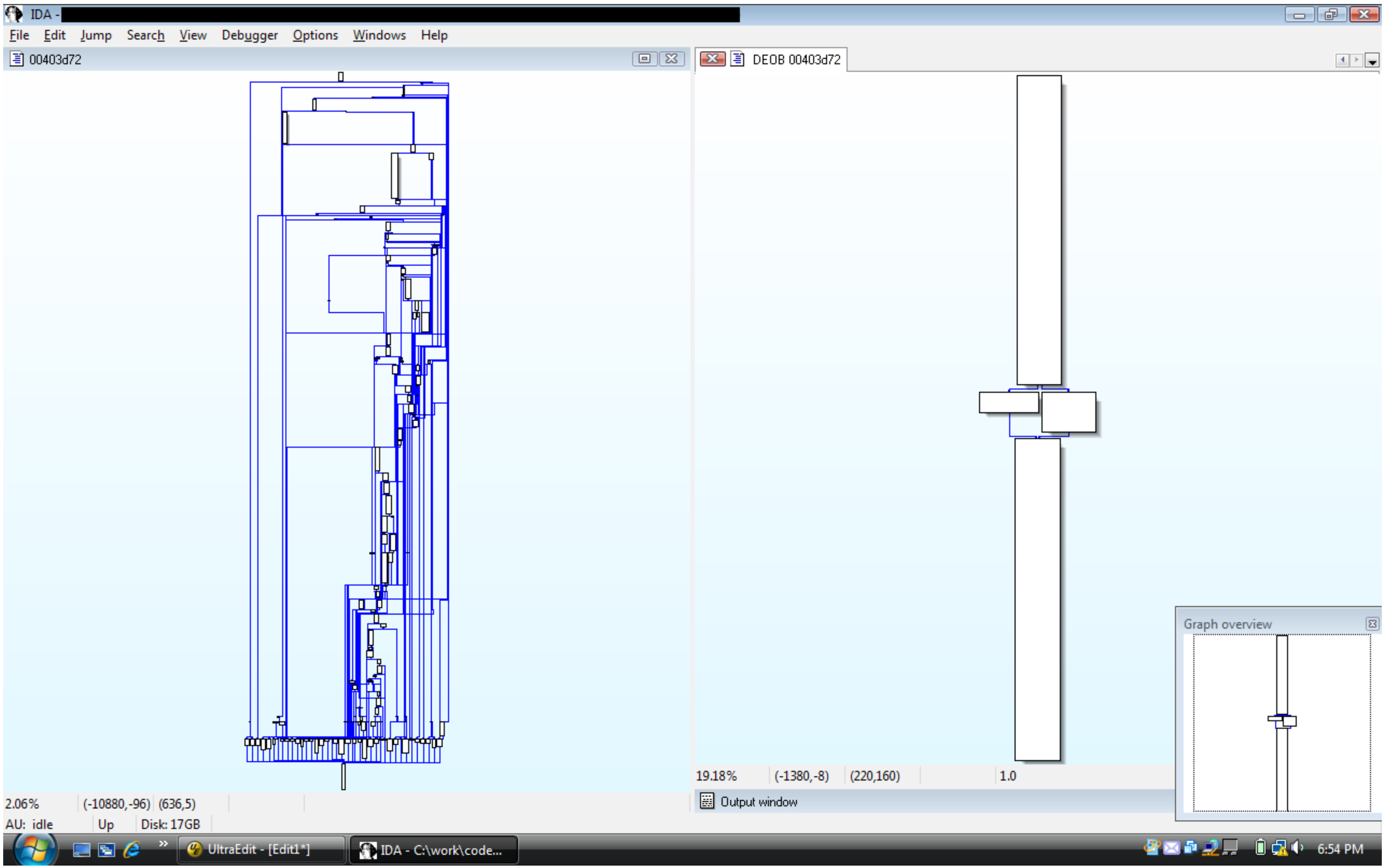
- Use abstract interpretation to remove superfluous basic blocks from control flow graphs.
- Abstract domain: three-valued bitvectors.

Anti-Tracing Control Obfuscation

```
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], 0FFFFFFBFh
or      [esp+0], edx
popf
jz      loc_34EC49
```

- 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.

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.

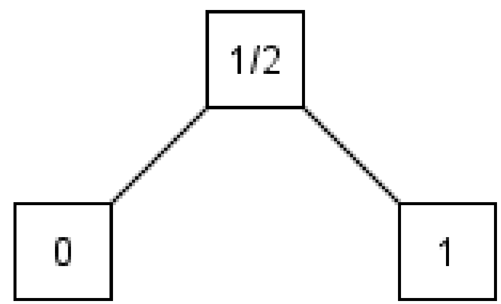
Problem: Unknown Bits

- 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?

	?	?	?	1	?	?	?	0
*	1	?	?	?	?	1	?	?
=	?	?	?	?	?	?	?	?

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

AND	0	1
0	0	0
1	0	1

- Standard concrete semantics for AND:
- What happens when we introduce ½ bits?
- ½ AND 0 = 0 AND ½ = 0 (0 AND anything = 0)
- ½ AND 1 = 1 AND ½ = …
  - If ½ = 0, then 0 AND 1 = 0
  - If ½ = 1, then 1 AND 1 = 1
  - Conflictory, therefore ½ AND 1 = ½.
  - Similarly ½ AND ½ = ½.
  - Final three-valued truth table:

AND	0	½	1
0	0	0	0
½	0	½	½
1	0	½	1

Abstract Semantics: Bitwise Operators

AND	0	½	1
0	0	0	0
½	0	½	½
1	0	½	1

OR	0	½	1
0	0	½	1
½	½	½	1
1	1	1	1

XOR	0	½	1
0	0	½	1
½	½	½	½
1	1	½	0

NOT	0	½	1
	1	½	0

These operators follow the same pattern as the derivation on the previous slide, and work exactly how you would expect

Abstract Semantics: Shift Operators

½	0	1	½	0	1	½	0	Some three-valued bitvector, call it BV
0	½	0	1	½	0	1	½	BV SHR 1
0	1	½	0	1	½	0	0	BV SHL 1
½	½	0	1	½	0	1	½	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.

Carry-Out	0	1	1	1	1	0	0	0
A[i]	0	1	0	1	1	0	1	0
B[i]	0	1	1	0	1	1	0	0
Carry-In	1	1	1	1	0	0	0	0
C[i]	1	1	0	0	0	1	1	0

- 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.

Carry-Out	0	0	0	½	½	½
A[i]	0	0	0	½	½	½
B[i]	0	0	0	½	½	½
Carry-In	0	0	½	½	½	0
Result	0	0	½	½	½	½

- 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

- Neg(x) = Not(x)+1
- Sub(x,y) = Add(x,~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 * 0x123
- 0x123 = 0001 0010 0011 = $2^8 + 2^5 + 2^1 + 2^0$
- B = A * $(2^8 + 2^5 + 2^1 + 2^0)$ (substitution)
- B = A * 2^8 + A * 2^5 + A * 2^1 + A * 2^0 (distributivity: * 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.

A	½	1	½	½	½	0	½	½
B	½	0	½	½	½	0	½	½

- For A < B, compute B-A and take the carry-out as the result
- For A <= B, compute (A < B) | (A == B).

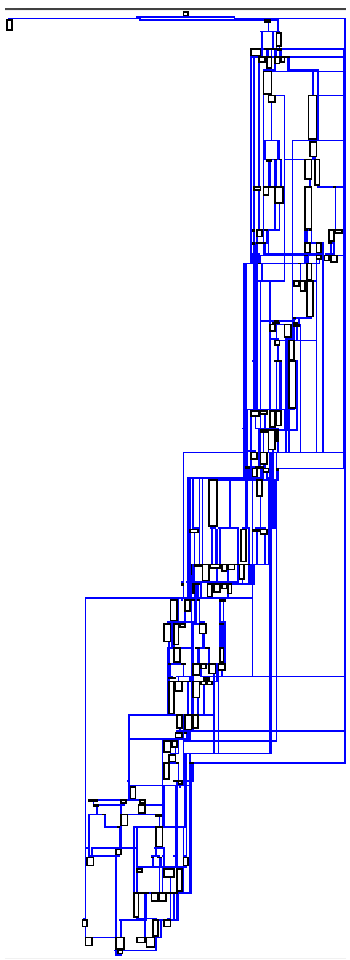
Deobfuscation Procedure

- Generate control flow graph

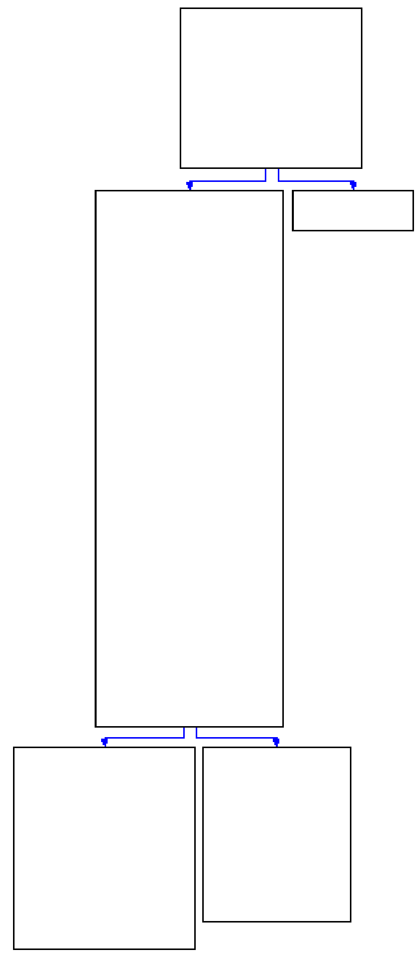
1. Apply the analysis to each basic block
2. If any conditional jump becomes unconditional, remove the false edge from the graph
3. Prune all vertices with no incoming edges (DFS)
4. Merge all vertices with a sole successor, whose successor has a sole predecessor
5. Iterate back to #1 until the graph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

- Stupid algorithm, could be majorly improved

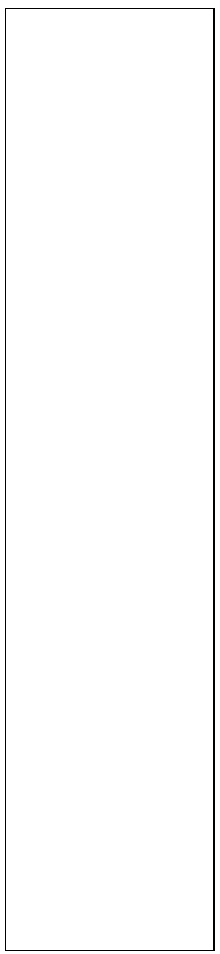
Progressive Deobfuscation



Original graph: 232 vertices



Deobfuscation round #1: five vertices



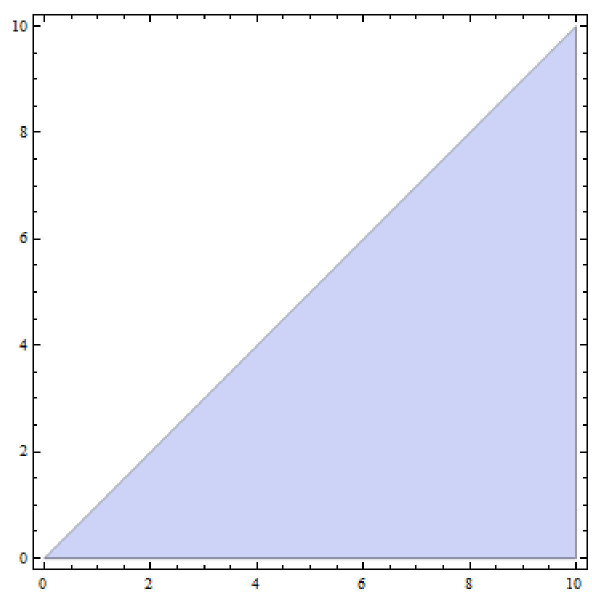
Deobfuscation round #2, final: one vertex

Example: Tracking ESP

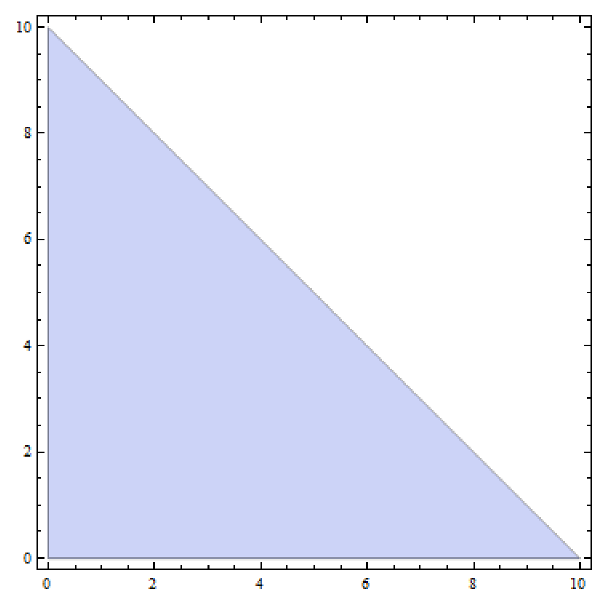
- We explore and generalize Ilfak's work on stack tracking.
- Abstract domains: convex polyhedra and friends in the relational domain family.

Concept: Relational Abstractions

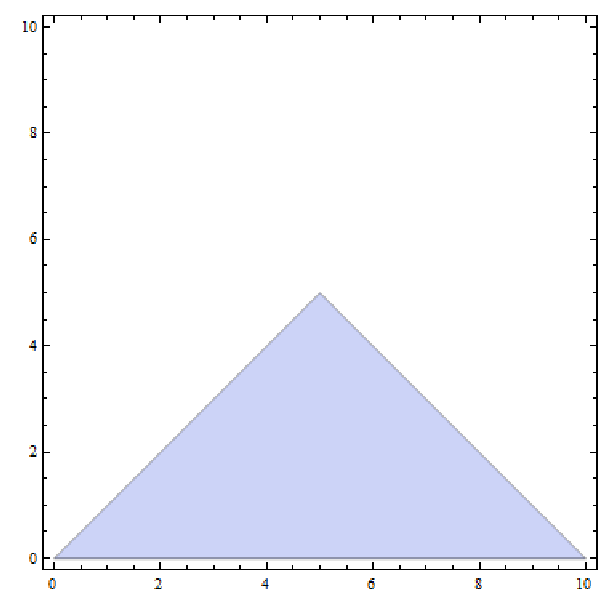
- So far, the analyses treated variables separately; we now consider analyses that treat variables in combination
- Below: two-dimensional convex polyhedra induced by linear inequalities over x and y



y < x



y < 10 - x



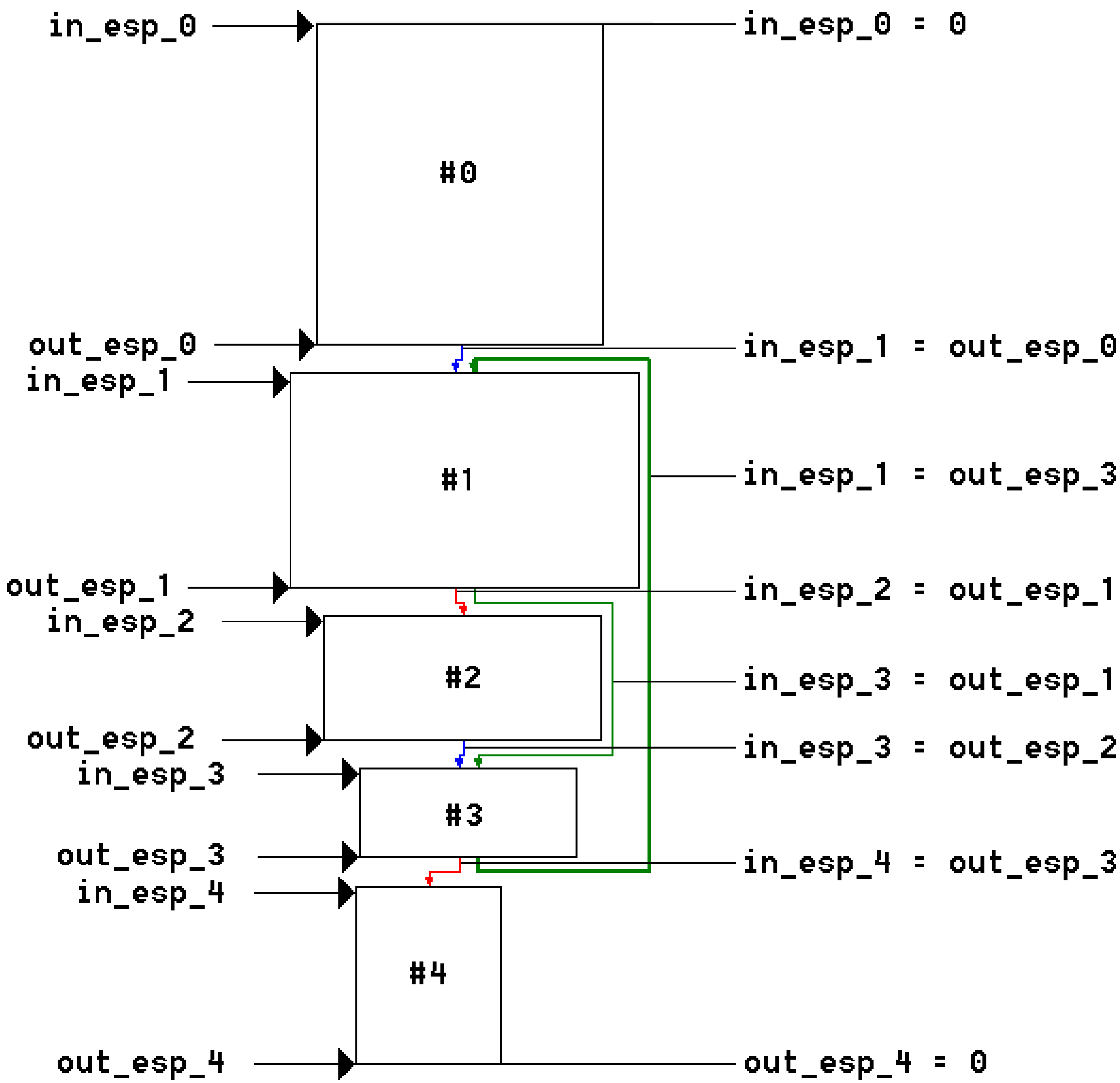
y < x && y < 10 - x && y > 0

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.

Stack Tracking: Inequalities

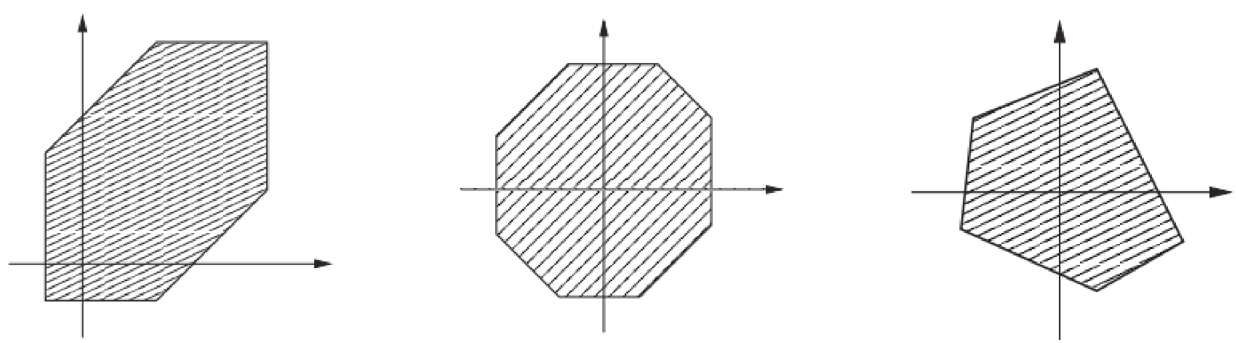
```
lea     ecx, [esp+0C4h+var_A8]
push    ecx
push    ebx
push    ebx
push    1012h
push    offset off_546AD8
push    eax
call    edx
mov     eax, [esi+4]
```

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:

$$out\_esp\_5 - in\_esp\_5 <= 24$$

Alternative Formulations

- Ilfak's solution uses polyhedra, which is potentially computationally expensive
- Note: all equations are of the form $v_i - v_j <= c_{ij}$, which can be solved in O(|V|*|E|) time with Bellman-Ford (or other PTIME solutions)



Zones $X_i - X_j \leq c_{ij}$

Octagons $\pm X_i \pm X_j \leq c_{ij}$

TVPLI $\alpha_{ij} X_i + \beta_{ij} X_j \leq c_{ij}$

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 = 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],Odd) = ([1,5],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, 0BBBBBBBh
add eax, ebx
```

IR to SMT Formula

```
T169b = cast(low,TypeReg_8,EBX)
T170b = cast(low,TypeReg_8,EBX)
T171b = T169b-T170b
EBX =
(EBX
& const(TypeReg_32,0xFFFFFF00)) |
  cast(unsigned,TypeReg_32,T171b)
label_010031FA:
; movzx ebx, bl
EBX =
  cast(unsigned,TypeReg_32,
  cast(low,TypeReg_8,EBX))
label_010031FD:
; add ebx, BBBBBBBBh
T172d = EBX
T173d = const(TypeReg_32,0xBBBBBBBB)
T174d = T172d+T173d
EBX = T174d
```

Part of the IR translation of the x86 snippet given on the previous slide.

```
assert(T169b = extract(7,0,EBX));
assert(T170b = extract(7,0,EBX));
assert(T171b = bvsub(T169b,T170b));
assert(EBX =
  bvor(
    bvand(EBX,mk_numeral(0xFFFFFF00)),
    mk_sign_ext(24,T171b)));
assert(EBX =
  mk_zero_ext(24,extract(7,0,EBX));
assert(T172d = EBX);
assert(T173d = mk_numeral(0xBBBBBBBB));
assert(T174d = bvadd(T172d,T173d));
assert(EBX = T174d);
```

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)*

```
sat
T180d -> bv3149642683[32]
T169b -> bv51[8]
T172d -> bv0[32]
T185bit -> bv0[1]
EBX -> bv51[32]
T170b -> bv51[8]
T179d -> bv0[32]
T175d -> bv1450744509[32]
T176d -> bv3149642683[32]
T173d -> bv3149642683[32]
T182bit -> bv1[1]
T177d -> bv305419896[32]
T178d -> bv0[32]
T184bit -> bv1[1]
T171b -> bv0[8]
T186bit -> bv1[1]
T174d -> bv3149642683[32]
T183bit -> bv0[1]
T181bit -> bv0[1]
T187d -> bv305419896[32]
EAX -> bv1450744509[32]
```

- 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

```
  mov   ecx, 20h
  mov   esi, offset a_ActivationCode
  lea   edi, [ebp+String_derived]
  mov   edx, [ebp+arg_0_serial_dw_1]
  mov   ebx, [ebp+arg_4_serial_dw_2]

loc_401105:
  lodsb
  sub   al, bl
  xor   al, dl
  stosb                                  x 32
  rol   edx, 1
  rol   ebx, 1
  loop  loc_401105
  mov   byte ptr [edi], 0
  push  offset a0how4zdy81jpe5xfu92kar
  lea   eax, [ebp+String_derived]
  push  eax
  call  lstrcmpA
```

- As before, generate an execution trace (statically) and convert to IR. Then convert the IR to an SMT formula.
- **Precondition:**
  a_ActivationCode[0] = X &&
  a_ActivationCode[1] = Y &&
  a_ActivationCode[2] = Z …
  where
  X = regcode[0],
  Y = regcode[1],
  Z = regcode[2], …
- **Postcondition**:
  String_derived[0] = '0' &&
  String_derived[1] = 'h' &&
  String_derived[2] = 'o' …

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**, naïvely. Count the number of one-bits set.

```
for(uint i = 1; i; i <<= 1)
  count += (val & i) != 0;
```

Iterative bit-tests

```
c00 = val & 0x00000001 ?     1 :   0;
c01 = val & 0x00000002 ? c00+1 : c00;
/* ... */
c31 = val & 0x80000000 ? c30+1 : c30;
```

Sequential ternary operator

Population Count via Bit Hacks

```
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, 0FFFFh
shr ebx, 10h
and ebx, 0FFFFh
add ebx, eax
mov eax, ebx
```

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

8-Bit Population Count via Bit Hacks

Round #1

	a	b	c	d	e	f	g	h
&	1	0	1	0	1	0	1	0
>>1	0	a	0	c	0	e	0	g

	a	b	c	d	e	f	g	h
&	0	1	0	1	0	1	0	1
	0	b	0	d	0	f	0	h

	0	a	0	c	0	e	0	g
+	0	b	0	d	0	f	0	h
=	**i**	**i**	**j**	**j**	**k**	**k**	**l**	**l**

Where
ii = a+b
jj = c+d
kk = e+f
ll = g+h

Round #2

	i	i	j	j	k	k	l	l
&	1	1	0	0	1	1	0	0
>>2	0	0	i	i	0	0	k	k

	i	i	j	j	k	k	l	l
&	0	0	1	1	0	0	1	1
	0	0	j	j	0	0	l	l

	0	0	i	i	0	0	k	k
+	0	0	j	j	0	0	l	l
=	**m**	**m**	**m**	**m**	**n**	**n**	**n**	**n**

Where
mmmm = ii+jj
nnnn = kk+ll

Round #3

	m	m	m	m	n	n	n	n
&	1	1	1	1	0	0	0	0
>>4	0	0	0	0	m	m	m	m

	m	m	m	m	n	n	n	n
&	0	0	0	0	1	1	1	1
	0	0	0	0	n	n	n	n

	0	0	0	0	m	m	m	m
+	0	0	0	0	n	n	n	n
=	**p**	**p**	**p**	**p**	**p**	**p**	**p**	**p**

Where
pppppppp = mmmm+nnnn
= ii+jj+kk+ll
= a+b+c+d+e+f+g+h

This is the population count.

Equivalence of Naïve and Bit Hack

```
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, 0FFFFh
shr ebx, 10h
and ebx, 0FFFFh
add ebx, eax
mov eax, ebx
```

```
c00 = val & 0x00000001 ?     1 :   0;
c01 = val & 0x00000002 ? c00+1 : c00;
/* ... */
c31 = val & 0x80000000 ? c30+1 : c30;
```

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)

```
lodsb byte 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, 00000004h
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 bl, 2Fh
xor bl, ch
pop ecx
add bl, 52h
sub bl, FCh
add al, bl
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:[esp], ebx
mov bl, 11h
inc bl
add bl, 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
and ah, C9h
push ebx
```

Is this ... (2 of 2)

```
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
shl 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?

```
lodsb 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
```

Obfuscated version of inc dword handler.

```
pop eax
inc dword ptr ss:[esp]
pushfd
```

Deobfuscated 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

```
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
```

Obfuscated version of sar dword handler.

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

Deobfuscated handler.

The sar instruction does not change the flags if the shiftand 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 $\mathcal{T}^{Ph} : \wp(\mathbb{P}) \to \wp(\mathbb{P})$ *is given by the point-wise extension of:*

$$\mathcal{T}^{Ph}(P_0) = \left\{ P_l \;\middle|\; \begin{array}{l} P_l = (m_l, a_l), \sigma = \sigma_0 \dots \sigma_{l-1}\sigma_l \in \mathbf{S}[\![P_0]\!], \sigma_l = \langle a_l, m_l, \theta_l, \Im_l \rangle, \\ (\sigma_{l-1}, \sigma_l) \in MT(P_0), \forall i \in [0, l-1[: (\sigma_i, \sigma_{i+1}) \notin MT(P_0) \end{array} \right\}$$

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

Theorem 1 $lfp^{\subseteq}\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!] = \mathbf{S}^{Ph}[\![P_0]\!]$.

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

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 $\alpha_{Ph} : \wp(\Sigma^*) \to \wp(\mathbb{P}^*)$ and $\gamma_{Ph} : \wp(\mathbb{P}^*) \to \wp(\Sigma^*)$, 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 $\sigma = \langle a_0, m_0, \theta_0, \Im_0 \rangle \dots \sigma_{i-1}\sigma_i \dots \sigma_n$ we define α_{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
  - https://www.openrce.org/blog/view/1963/
- 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