Analysis of x86 Executables using Abstract Interpretation

From Theory to Practice

Henny Sipma

Kestrel Technology, LLC

International Static Analysis Symposium

Saint Malo, September 9, 2015
Background and Experience

Process control engineer (The Hague, Singapore, Houston) 7 years

PhD, research associate with Prof. Zohar Manna in formal verification (Stanford University, Palo Alto) 14 years

Static analysis engineer (Kestrel Technology, Palo Alto)
- C source code
- Java byte code
- x86 executables 8 years
Kestrel Technology

Founded: 2000
Location: Palo Alto, California
Core activity: Sound Static Analysis of Software
Languages supported: C source, Java bytecode, x86 executables
Underlying technology: Abstract interpretation (Cousot & Cousot, 1977)
The CodeHawk Tool Suite

- **Java byte code front end**
  - Sound abstraction from Java byte code into CHIF
- **C source code front end**
  - Sound abstraction from preprocessed CIL code into CHIF
- **x86 binary front end**
  - Disassembly abstraction from x86 binary code into CHIF

Abstract interpretation engine

- Iterators
- Abstract domains:
  - Constants
  - Intervals
  - Strided intervals
  - Linear equalities
  - Polyhedra
  - Symbolic sets
  - Value sets
  - Taint
Outline

- Binary Analyzer
  - architecture
  - disassembler
  - abstraction
  - analysis
- Test and Evaluation
  - corpus
  - infrastructure
  - metrics
  - results
- Use Cases
  - reverse engineering
  - vulnerability research
  - malware analysis
- Conclusions
The CodeHawk Binary Analyzer

32-bit PE

| .exe | .dll |

x86 front end

Targeted at

- vulnerability researchers
- reverse engineers
- software assurance centers
- malware analysis/forensics
- .........

Complementary to

- IDA Pro
- objdump
- ......

Abstract domains:
- constants
- intervals
- linear equalities
- polyhedra
- value sets
- symbolic sets

Goal: produce a commercial product
CodeHawk Binary Analyzer: Architecture

1. **.dll** and **.exe**
   - Disassemble and construct functions
   - Data blocks, function names, function entry points

2. Translate to CHIF
   - Function summaries (xml)
   - Library function summaries
   - Indirect jump targets

3. **analyze CHIF**
   - Function invariants (xml)

Diagram shows the flow of processes from input files to output analyses, highlighting key components such as disassembly, translation to CHIF, and analysis of invariants.
CodeHawk Binary Analyzer: Disassembler

- .dll
- .exe

- disassemble and construct functions
- function summaries (xml)
- translate to CHIF
- analyze CHIF

- data blocks
- function names
- function entry points
- library function summaries
- indirect jump targets
- invariants
- function invariants (xml)
CodeHawk Binary Analyzer: Disassembler

32-bit PE

- .exe
- .dll

**disassemble:**
- read headers
- identify jump tables
- exclude data blocks
- disassemble code sections

**construct functions**

**data blocks**

Disassembly method: linear sweep
Recognizes ~900 opcodes (out of ~1700), including SSE and AVX instructions

~ 160 internal instruction types:
- Add
- Mov
- Push
- Pop
- Jcc
- ...
Fragment from nginx.exe

B 0x570fe9  66 0f ef c0
0x570fed  51
0x570fee  53
0x570fef  8b c1
0x570ff1  83 e0 0f
0x570ff4  85 c0
0x570ff6  75 7f
B 0x570ff8  8b c2
0x570ffa  83 e2 7f
0x570ffd  c1 e8 07
0x571000  74 37
B 0x571002  8d a4 24 00 00 00 00 00
B 0x571009  66 0f 7f 01
0x57100d  66 0f 7f 41 10
0x571012  66 0f 7f 41 20
0x571017  66 0f 7f 41 30
0x57101c  66 0f 7f 41 40
0x571021  66 0f 7f 41 50
0x571026  66 0f 7f 41 60
0x57102b  66 0f 7f 41 70
0x571030  8d 89 80 00 00 00 00
0x571036  48
0x571037  75 d0
B 0x571039  85 d2
0x57103b  74 37

pxor %xmm0, %xmm0
push ecx
push edx
mov eax, ecx
and eax, $0xf
test eax, eax
jnz 0x571077
mov eax, edx
and edx, $0x7f
shr eax, $0x7
jz 0x571039
lea esp, 0x0(%esp,,1)

movdqa (%ecx), %xmm0
movdqa 0x10(%ecx), %xmm0
movdqa 0x20(%ecx), %xmm0
movdqa 0x30(%ecx), %xmm0
movdqa 0x40(%ecx), %xmm0
movdqa 0x50(%ecx), %xmm0
movdqa 0x60(%ecx), %xmm0
movdqa 0x70(%ecx), %xmm0
lea ecx, 0x80(%ecx)
dec eax
jnz 0x571009

test edx, edx
jz 0x571074
CodeHawk Binary Analyzer: Disassembler

32-bit PE

exe

dll

data blocks

disassemble:
read headers
identify jump tables
exclude data blocks
disassemble code sections

construct functions

~ 160 internal instruction types:
Add
Mov
Push
Pop
Jcc
...

Disassembly method: linear sweep
Recognizes ~900 opcodes (out of ~1700),
including SSE and AVX instructions
CodeHawk Binary Analyzer: Construct Functions

1. Collect direct call targets, combine with user-provided function entry points
2. For every function entry point:
   1. Identify non-returning function calls
   2. Identify basic blocks
   3. Construct control flow graph
   4. Connect conditional jumps with test expressions
   5. Identify function call arguments (using library function summaries, if available)
Connect conditional jumps with test instructions

Flags used by condition codes

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Condition Code</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>CcOverflow</td>
<td>CcNotOverflow</td>
<td>[ OFlag ]</td>
</tr>
<tr>
<td>CcCarry</td>
<td>CcNotCarry</td>
<td>[ CFlag ]</td>
</tr>
<tr>
<td>CcZero</td>
<td>CcNotZero</td>
<td>[ ZFlag ]</td>
</tr>
<tr>
<td>CcBelowEqual</td>
<td>CcAbove</td>
<td>[ CFlag ; ZFlag ]</td>
</tr>
<tr>
<td>CcSign</td>
<td>CcNotSign</td>
<td>[ SFlag ]</td>
</tr>
<tr>
<td>CcParityEven</td>
<td>CcParityOdd</td>
<td>[ PFlag ]</td>
</tr>
<tr>
<td>CcLess</td>
<td>CcGreaterEqual</td>
<td>[ SFlag ; OFlag ]</td>
</tr>
<tr>
<td>CcLessEqual</td>
<td>CcGreater</td>
<td>[ ZFlag ; SFlag ; OFlag ]</td>
</tr>
</tbody>
</table>

Flags set by various instructions

- Add: [ OFlag ; CFlag ; ZFlag ; PFlag ; SFlag ]
- BitScanForward (bsf): [ ZFlag ]
- BitTestComplement (btc): [ OFlag ; CFlag ; PFlag ; SFlag ]
- Cmp: [ OFlag ; CFlag ; ZFlag ; PFlag ; SFlag ]
- Decrement: [ OFlag ; ZFlag ; PFlag ; SFlag ]
- ...........

.............
Fragment from nginx-1.2.7

B 0x47e9a1 83 f8 08
  0x47e9a4 75 24
B 0x47e9a6 83 bf 8c 00 00 00 01
  0x47e9ad 75 1b
B 0x47e9af 8b 56 04
  0x47e9b2 50
  0x47e9b3 68 74 dd 5f 00
  0x47e9b8 52
  0x47e9b9 e8 95 28 f8 ff
  0x47e9be 83 c4 0c
  0x47e9c1 85 c0
  0x47e9c3 b8 04 00 00 00
  0x47e9c8 74 05
B 0x47e9ca b8 14 00 00 00
B 0x47e9cf 5e
  0x47e9d0 5f
  0x47e9d1 5b
  0x47e9d2 c3

cmp eax, $0x8
jnz 0x47e9ca
jmp not zero

jmp not zero

cmp 0x8c(%edi), $0x1
jnz 0x47e9ca
mov edx, 0x4(%esi)
push eax
push $0x5fdd74
push edx
call 0x401253
add esp, $0xc
test eax, eax
mov eax, $0x4
jz 0x47e9cf
jmp zero

mov eax, $0x14
pop esi
pop edi
pop ebx
ret
1. Collect direct call targets, combine with user-provided function entry points
2. For every function entry point:
   1. Identify non-returning function calls
   2. Identify basic blocks
   3. Construct control flow graph
   4. Connect conditional jumps with test expressions
   5. Identify function call arguments (using library function summaries, if available)
Connect function calls with their arguments (non-gcc)
CodeHawk Binary Analyzer: Abstraction

- .dll
- .exe
- disassemble and construct functions
- function summaries (xml)
- translate to CHIF
- analyze CHIF
- invariants
- indirect jump targets
- data blocks
- function names
- function entry points
- library function summaries
- function invariants (xml)
CHIF: CodeHawk Internal Form

provides:
- imperative, register-based language
- programming-language independent
- structured, hierarchical control flow graph
- types: integer, symbolic, array, struct
- basic arithmetic operations: +, -, *, /
- basic set operations: union, intersection, difference
- predicates: <, <=, >, >=, =, !=, element-of

does not have:
- pointers
- bitwise operations
- floating point operations

but allows:
- "arbitrary" user-defined semantics
Translation into CHIF

Abstraction into CHIF

Requirement: Over-approximating semantics

All behaviors of the assembly function must be included in the set of behaviors of the CHIF function

but not necessarily the other way around
Abstraction into CHIF

Provide abstract semantics for all ~160 instruction types

Some are precise:

Add (op1,op2)  \[ \text{op1} := \text{op1} + \text{op2} \]
Mov (op1,op2)  \[ \text{op1} := \text{op2} \]
Push op  \[ \text{esp} := \text{esp} - 4 \; ; \; \text{mem} \lbrack \text{esp} \rbrack := \text{op} \]

Some involve limited non-determinism:

AddCarry (op1,op2)  \[ \text{op1} := \text{op1} + \text{op2} \quad \text{or} \quad \text{op1} := \text{op1} + \text{op2} + 1 \]

Some abstract completely:

PackedAlignRight (op,.,.)  \[ \text{op} := \text{TOP} \quad \text{(destination is abstracted)} \]
Abstraction into CHIF

Some are complex:

RepMovs (width, dst, src)

(*---------------------------------------------*
 * RepMovs: Move ECX bytes/words/doublewords from DS:[ESI] to ES:[EDI] *
 * Semantics (parallel) *
 *   let size = ECX * width in *
 *   ECX := 0 *
 *   if DF = 0 : *
 *     ESI := ESI + size *
 *     EDI := EDI + size *
 *     mem [ EDI ; EDI + size - width ] := mem [ ESI ; ESI + size - width ] *
 *   if DF = 1 : *
 *     ESI := ESI - size *
 *     EDI := EDI - size *
 *     mem [ EDI - size + width ; EDI ] := mem [ ESI - size + width ; ESI ] *
 * --------------------------------------------- *)
Abstraction into CHIF

PROBLEM: THERE ARE NO VARIABLES

\texttt{mov 0xc(eax), ecx}

assign the contents of register ecx to the memory location pointed to by the contents of register eax plus twelve

We have to know the value of eax before we can abstract this instruction

APPROACH

Incremental creation of variables by iterative analysis
Abstraction into CHIF

- .dll
- .exe

- data blocks
- function names
- function entry points

- disassemble and construct functions

- translate to CHIF

- analyze CHIF

- function summaries (xml)
- function invariants (xml)

- library function summaries

- indirect jump targets
- invariants
Start with

- Eax
- Ebx
- Ecx
- Edx
- Esp
- Ebp
- Esi
- Edi

**generic memory location: L**

```
mov 0xc(eax), ecx        L := ecx
mov ecx, 0xc(eax)        ecx := TOP
```

and similar for the other 150 instruction types
.. and incrementally work towards (for each function)

Eax  Ebx  Ecx  Edx  Esp  Ebp  Esi  Edi

**generic memory location: L**

global data

```
...  ...
```

local stack frame

```

ereturn-address
arg-1  arg-2
```

esp-in
Analysis: Resolve Memory References

Initially:  Esp = Esp_in

with Esp_in the address of the return-address

with invariant  Eax = Esp_in - 4

mov -0xc(eax),ecx  resolves to  var.0016 := Ecx

mov 0xc(eax),ecx  resolves to  arg.0008 := Ecx
.. and incrementally work towards (for each function)

Eax  Ebx  Ecx  Edx  Esp  Ebp  Esi  Edi

generic memory location: L

global data

heap-allocated regions

local stack frame

arg-2
arg-1
return-address
function seedify

```
0x4014fb  push ebp                esp := esp - 4 ;  L? := ebp
0x4014fc  mov ebp, esp             ebp := esp
0x4014fe  push ebx                esp := esp - 68 ;  L? := ebx
0x4014ff  sub esp, $0x44           edx := TOP
0x401502  mov -0x12(ebp), $0x1      eax := TOP
0x401508  movsx edx, -0x12(ebp)    eax := eax * edx
0x40150c  movsx eax, -0x12(ebp)    L? := 1
0x401510  imul eax, eax, edx      eax := eax * 4
0x401513  mov -0x38(ebp,eax,4), $0x1 [ strcpy:src = eax ]
0x40151c  mov eax, -0x34(ebp)      eax := eax + TOP
0x40151f  shl eax, $0x2           eax := TOP
0x401522  add eax, 0x8(ebp)       eax := ebp - 43
0x401525  mov eax, (eax)          [ strcpy:dst = eax ]
0x401527  mov 0x4(esp,,1), eax     call strcpy
0x40152b  lea eax, -0x2b(ebp)      call seedify
0x40152e  mov 0x0(esp,,1), eax     [ strcpy:dst = eax ]
0x401531  call 0x407020
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
<th>invariants</th>
<th>2nd translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4014fb</td>
<td>push ebp</td>
<td>esp = esp0</td>
<td>esp := esp - 4 ; var.0004 := ebp</td>
</tr>
<tr>
<td>0x4014fc</td>
<td>mov ebp, esp</td>
<td>esp = esp0 - 4</td>
<td>ebp := esp</td>
</tr>
<tr>
<td>0x4014fe</td>
<td>push ebx</td>
<td>ebp = esp0 - 4</td>
<td>esp := esp - 4 ; var.0008 := ebx</td>
</tr>
<tr>
<td>0x4014ff</td>
<td>sub esp, $0x44</td>
<td>esp = esp0 - 8</td>
<td>esp := esp - 68</td>
</tr>
<tr>
<td>0x401502</td>
<td>mov -0x12(ebp), $0x1</td>
<td>ebp = esp0 - 4</td>
<td>var.0022 := 1</td>
</tr>
<tr>
<td>0x401508</td>
<td>movsx edx, -0x12(ebp)</td>
<td>ebp = esp0 - 4</td>
<td>edx := var.0022</td>
</tr>
<tr>
<td>0x40150c</td>
<td>movsx eax, -0x12(ebp)</td>
<td>ebp = esp0 - 4</td>
<td>eax := var.0022</td>
</tr>
<tr>
<td>0x401510</td>
<td>imul eax, eax, edx</td>
<td>eax := eax * edx</td>
<td>eax := eax * edx</td>
</tr>
<tr>
<td>0x401513</td>
<td>mov -0x38(ebp,eax,4), $0x1</td>
<td>ebp = esp0 - 4</td>
<td>L? := 1</td>
</tr>
<tr>
<td>0x40151c</td>
<td>mov eax, -0x34(ebp)</td>
<td>ebp = esp0 - 4</td>
<td>eax := var.0056</td>
</tr>
<tr>
<td>0x40151f</td>
<td>shl eax, $0x2</td>
<td>ebp = esp0 - 4</td>
<td>eax := eax * 4</td>
</tr>
<tr>
<td>0x401522</td>
<td>add eax, 0x8(ebp)</td>
<td>ebp = esp0 - 4</td>
<td>eax := eax + arg.0004</td>
</tr>
<tr>
<td>0x401525</td>
<td>mov eax, (eax)</td>
<td>ebp = esp0 - 4</td>
<td>eax := TOP</td>
</tr>
<tr>
<td>0x401527</td>
<td>mov 0x4 esp, eax</td>
<td>esp = esp0 - 76</td>
<td>[ strcpy:src := eax ]</td>
</tr>
<tr>
<td>0x40152b</td>
<td>lea eax, -0x2b(ebp)</td>
<td>ebp = esp0 - 4</td>
<td>eax := esp0 - 47</td>
</tr>
<tr>
<td>0x40152e</td>
<td>mov 0x0 esp, eax</td>
<td>esp = esp0 - 76</td>
<td>[ strcpy:dst := eax ]</td>
</tr>
<tr>
<td>0x401531</td>
<td>call 0x407020</td>
<td></td>
<td>call strcpy</td>
</tr>
<tr>
<td>Address</td>
<td>Instruction</td>
<td>Invariants</td>
<td>3rd Translation</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------</td>
<td>---------------------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x4014fb</td>
<td>push ebp</td>
<td></td>
<td>esp := esp0</td>
</tr>
<tr>
<td>0x4014fc</td>
<td>mov ebp, esp</td>
<td></td>
<td>ebp := esp</td>
</tr>
<tr>
<td>0x4014fe</td>
<td>push ebx</td>
<td></td>
<td>esp := esp0 – 4</td>
</tr>
<tr>
<td>0x4014ff</td>
<td>sub esp, $0x44</td>
<td></td>
<td>esp := esp0 – 4</td>
</tr>
<tr>
<td>0x401502</td>
<td>mov -0x12(ebp), $0x1</td>
<td></td>
<td>ebp := esp0 – 4</td>
</tr>
<tr>
<td>0x401508</td>
<td>movsx edx, -0x12(ebp)</td>
<td></td>
<td>ebp := esp0 – 4</td>
</tr>
<tr>
<td>0x40150c</td>
<td>movsx eax, -0x12(ebp)</td>
<td></td>
<td>ebp := esp0 – 4</td>
</tr>
<tr>
<td>0x401510</td>
<td>imul eax, eax, edx</td>
<td>eax = edx = 1</td>
<td>ebx := var.0022</td>
</tr>
<tr>
<td>0x401513</td>
<td>mov -0x38(ebp,eax,4), $0x1</td>
<td></td>
<td>eax := var.0022</td>
</tr>
<tr>
<td>0x40151c</td>
<td>mov eax, -0x34(ebp)</td>
<td></td>
<td>eax := var.0056</td>
</tr>
<tr>
<td>0x40151f</td>
<td>shl eax, $0x2</td>
<td></td>
<td>eax := eax + 4</td>
</tr>
<tr>
<td>0x401522</td>
<td>add eax, 0x8(ebp)</td>
<td></td>
<td>eax := TOP</td>
</tr>
<tr>
<td>0x401525</td>
<td>mov eax, (eax)</td>
<td></td>
<td>esp := esp0 – 76</td>
</tr>
<tr>
<td>0x401527</td>
<td>mov 0x4(eesp,,1), eax</td>
<td></td>
<td>[ strcpy:src := eax ]</td>
</tr>
<tr>
<td>0x40152b</td>
<td>lea eax, -0x2b(ebp)</td>
<td></td>
<td>eax := esp0 – 4</td>
</tr>
<tr>
<td>0x40152e</td>
<td>mov 0x0(eesp,,1), eax</td>
<td></td>
<td>esp := esp0 – 76</td>
</tr>
<tr>
<td>0x401531</td>
<td>call 0x407020</td>
<td></td>
<td>[ strcpy:dst := esp0-47 ]</td>
</tr>
</tbody>
</table>
0x4014fb  push ebp
0x4014fc  mov ebp, esp
0x4014fe  push ebx
0x4014ff  sub esp, $0x44
0x401502  mov -0x12(ebp), $0x1
0x401508  movsx edx, -0x12(ebp)
0x40150c  movsx eax, -0x12(ebp)
0x401510  imul eax, eax, edx
0x401513  mov -0x38(ebp,eax,4), $0x1
0x40151c  mov eax, -0x34(ebp)
0x40151f  shl eax, $0x2
0x401522  add eax, 0x8(ebp)
0x401525  mov eax, (eax)
0x401527  mov 0x4 esp,,1), eax
0x40152b  lea eax, -0x2b(ebp)
0x40152e  mov 0x0 esp,,1), eax
0x401531  call 0x407020

**invariants**

| esp = esp0 |
| esp = esp0 – 4 |
| ebp = esp0 – 4 |
| esp = esp0 – 8 |
| ebp = esp0 – 4 |
| ebp = esp0 – 4 |
| eax = edx = 1 |
| ebp = esp0 – 4, eax = 1 |
| ebp = esp0 – 4 |
| eax = 1 |
| ebp = esp0 – 4 |
| eax = 4 + arg.0004 |
| eax := (arg.0004)[4] |
| esp = esp0 – 76 |
| ebp = esp0 – 4 |
| esp = esp0 – 76 |

**4th translation**

<p>| esp := esp – 4 ; var.0004 := ebp |
| ebp := esp |
| esp := esp – 4 ; var.0008 := ebx |
| esp := esp - 68 |
| var.0022 := 1 |
| edx := var.0022 |
| eax := var.0022 |
| eax := eax * edx |
| var.0056 := 1 |
| eax := var.0056 |
| eax := eax * 4 |
| eax := 4 + arg.0004 |
| eax := (arg.0004)[4] |
| [ strcpy:src := (arg.004)[4] ] |
| esp = esp0 – 76 |
| ebp = esp0 – 4 |
| esp = esp0 – 76 |
| [ strcpy:dst := esp0 - 47 ] |
| call strcpy |</p>
<table>
<thead>
<tr>
<th>2nd translation</th>
<th>3rd translation</th>
<th>4th translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>... ; var.0004 := ebp</td>
<td>... ; var.0004 := ebp</td>
<td>... ; var.0004 := ebp</td>
</tr>
<tr>
<td>ebp := esp</td>
<td>ebp := esp</td>
<td>ebp := esp</td>
</tr>
<tr>
<td>... ; var.0008 := ebx</td>
<td>... ; var.0008 := ebx</td>
<td>... ; var.0008 := ebx</td>
</tr>
<tr>
<td>esp := esp - 68</td>
<td>esp := esp - 68</td>
<td>esp := esp - 68</td>
</tr>
<tr>
<td>var.0022 := 1</td>
<td>var.0022 := 1</td>
<td>var.0022 := 1</td>
</tr>
<tr>
<td>edx := var.0022</td>
<td>edx := var.0022</td>
<td>edx := var.0022</td>
</tr>
<tr>
<td>eax := var.0022</td>
<td>eax := var.0022</td>
<td>eax := var.0022</td>
</tr>
<tr>
<td>eax := eax * edx</td>
<td>eax := eax * edx</td>
<td>eax := eax * edx</td>
</tr>
<tr>
<td>L? := 1</td>
<td>var.0056 := 1</td>
<td>var.0056 := 1</td>
</tr>
<tr>
<td>eax := var.0056</td>
<td>eax := var.0056</td>
<td>eax := var.0056</td>
</tr>
<tr>
<td>eax := eax * 4</td>
<td>eax := eax * 4</td>
<td>eax := eax * 4</td>
</tr>
<tr>
<td>eax := eax + arg.0004</td>
<td>eax := eax + arg.0004</td>
<td>eax := 4 + arg.0004</td>
</tr>
<tr>
<td>eax := TOP</td>
<td>eax := TOP</td>
<td>eax := (arg.0004)[4]</td>
</tr>
<tr>
<td>[ strcpy:src := eax ]</td>
<td>[ strcpy:src := eax ]</td>
<td>[ strcpy:src := (arg.004)[4] ]</td>
</tr>
<tr>
<td>eax := esp0 – 47</td>
<td>eax := esp0 – 47</td>
<td>eax := esp0 – 47</td>
</tr>
<tr>
<td>[ strcpy:dst := eax ]</td>
<td>[ strcpy:dst := esp0-47 ]</td>
<td>[ strcpy:dst := esp0 - 47 ]</td>
</tr>
<tr>
<td>call strcpy</td>
<td>call strcpy</td>
<td>call strcpy</td>
</tr>
</tbody>
</table>
Abstraction into CHIF: Conditional Jumps

during disassembly:

..... connect conditional jumps with test (flag-setting) instruction

at translation:

• freeze variables involved in test instruction
• combine test and condition code into predicate on variables
Fragment from nginx-1.2.7

```
B 0x47e9a1  83 f8 08
  0x47e9a4  75 24
B 0x47e9a6  83 bf 8c 00 00 00 01
  0x47e9ad  75 1b
B 0x47e9af  8b 56 04
  0x47e9b2  50
  0x47e9b3  68 74 dd 5f 00
  0x47e9b8  52
  0x47e9b9  e8 95 28 f8 ff
  0x47e9be  83 c4 0c
  0x47e9c1  85 c0
  0x47e9c3  b8 04 00 00 00
  0x47e9c8  74 05
B 0x47e9ca  b8 14 00 00 00
B 0x47e9cf  5e
  0x47e9d0  5f
  0x47e9d1  5b
  0x47e9d2  c3
```

```assembly
cmp eax, $0x8
jnz 0x47e9ca

cmp 0x8c(%edi), $0x1
jnz 0x47e9ca

mov edx, 0x4(%esi)
push eax
push $0x5fdd74
push edx
call 0x401253
add esp, $0xc

```

test eax, eax
mov eax, $0x4
jz 0x47e9cf

mov eax, $0x14
pop esi
pop edi
pop ebx
ret
```
### Abstraction into CHIF: Conditional Jumps
(adapted from Balakrishnan, Reps)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Predicate</th>
<th>Condition</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>jc</td>
<td>$y &lt;_u x$</td>
<td>jz</td>
<td>$x = 1$</td>
</tr>
<tr>
<td>jnc</td>
<td>$y \geq _u x$</td>
<td>jnz</td>
<td>$x \neq 1$</td>
</tr>
<tr>
<td>jz</td>
<td>$x = y$</td>
<td>js</td>
<td>$x \leq 0$</td>
</tr>
<tr>
<td>jnz</td>
<td>$x \neq y$</td>
<td>jns</td>
<td>$x &gt; 0$</td>
</tr>
<tr>
<td>jbe</td>
<td>$y &lt;_u x$</td>
<td>jl</td>
<td>$x \leq 0$</td>
</tr>
<tr>
<td>ja</td>
<td>$y &gt;_u x$</td>
<td>jge</td>
<td>$x &gt; 0$</td>
</tr>
<tr>
<td>jl</td>
<td>$y &lt; x$</td>
<td>jle</td>
<td>$x \leq 1$</td>
</tr>
<tr>
<td>jge</td>
<td>$y \geq x$</td>
<td>jg</td>
<td>$x &gt; 1$</td>
</tr>
<tr>
<td>jle</td>
<td>$y \leq x$</td>
<td>js</td>
<td>$y &lt; x$</td>
</tr>
<tr>
<td>jg</td>
<td>$y &gt; x$</td>
<td>jns</td>
<td>$y \geq x$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Condition</th>
<th>Flags</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>jc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jnc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jz</td>
<td>ZF</td>
<td>$x = 0$</td>
</tr>
<tr>
<td>jnz</td>
<td>$\neg$ZF</td>
<td>$x \neq 0$</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>$x &lt; 0$</td>
</tr>
<tr>
<td>jns</td>
<td>$\neg$SF</td>
<td>$x \geq 0$</td>
</tr>
<tr>
<td>jbe</td>
<td>ZF</td>
<td>$x = 0$</td>
</tr>
<tr>
<td>ja</td>
<td>$\neg$ZF</td>
<td>$x \neq 0$</td>
</tr>
<tr>
<td>jl</td>
<td>SF</td>
<td>$x &lt; 0$</td>
</tr>
<tr>
<td>jge</td>
<td>$\neg$SF</td>
<td>$x \geq 0$</td>
</tr>
<tr>
<td>jle</td>
<td>ZF $\lor$ SF</td>
<td>$x \leq 0$</td>
</tr>
<tr>
<td>jg</td>
<td>$\neg$ZF $\land$ $\neg$SF</td>
<td>$x &gt; 0$</td>
</tr>
</tbody>
</table>
Fragment from nginx-1.2.7

```
0x47e9a1  83 f8 08
0x47e9a4  75 24
0x47e9a6  83 bf 8c 00 00 00 01
0x47e9ad  75 1b
0x47e9af  8b 56 04
0x47e9b2  50
0x47e9b3  68 74 dd 5f 00
0x47e9b8  52
0x47e9b9  e8 95 28 f8 ff
0x47e9be  83 c4 0c
0x47e9c1  85 c0
0x47e9c3  b8 04 00 00 00
0x47e9c8  74 05
```

cmp eax, $0x8
jnz 0x47e9ca

if (arg.0004[136])[0] != 8 then goto 0x47e9ca

cmp 0x8c(edi), $0x1
jnz 0x47e9ca

if arg.0004[140] != 1 then goto 0x47e9ca

test eax, eax,
mov eax, $0x4
jz 0x47e9cf

if 0x41253_rtn@0x47e9b9 = 0 then goto 0x47e9cf
Abstraction into CHIF: Conditional Jumps

```
cmp eax, $0x8
jnz 0x47e9ca

cmp 0x8c(edi), $0x1
jnz 0x47e9ca

test eax, eax,
mov eax, $0x4
jz 0x47e9cf
```

In all (3948) functions analyzed for nginx-1.2.7:

- **conditional jumps**: 31,404
- **conditional jumps with predicate**: 28,306 (90%)
- **conditional jumps with predicate on argument**: 8,603 (27%)
- **conditional jumps with predicate on function return value**: 6,948 (22%)
Abstraction into CHIF: Function calls

during disassembly:

..... connect move/push instructions with function arguments

at translation:

• freeze variables that provide the function arguments
• retrieve argument values at the location of the call

if the called function has a function summary:

• constrain return value (eax) according to postcondition
• apply side effects to corresponding arguments
• generate new heap base pointer if called function allocates memory
Function Summaries

Provide:

- Types of arguments and return value
- Preconditions (buffer-size, null-dereference, linear constraints)
- Post-conditions (constraints on return value)
- Side effects (memory writes through pointers, writes to global variables)
- Stack adjustment

Manually constructed for library functions

Automatically derived for application functions
# Library Function Summaries

<table>
<thead>
<tr>
<th>dll</th>
<th>function summaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>advapi32</td>
<td>179</td>
</tr>
<tr>
<td>comctl32</td>
<td>9</td>
</tr>
<tr>
<td>comdlg32</td>
<td>28</td>
</tr>
<tr>
<td>crypt32</td>
<td>3</td>
</tr>
<tr>
<td>dwmapi</td>
<td>5</td>
</tr>
<tr>
<td>gdi32</td>
<td>181</td>
</tr>
<tr>
<td>imm32</td>
<td>21</td>
</tr>
<tr>
<td>kernel32</td>
<td>518</td>
</tr>
<tr>
<td>msvcrt</td>
<td>147</td>
</tr>
<tr>
<td>msvfw32</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dll</th>
<th>function summaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>mswsock</td>
<td>2</td>
</tr>
<tr>
<td>netapi32</td>
<td>3</td>
</tr>
<tr>
<td>ole32</td>
<td>29</td>
</tr>
<tr>
<td>oleaut32</td>
<td>14</td>
</tr>
<tr>
<td>opengl32</td>
<td>4</td>
</tr>
<tr>
<td>psapi</td>
<td>2</td>
</tr>
<tr>
<td>secur32</td>
<td>5</td>
</tr>
<tr>
<td>shell32</td>
<td>20</td>
</tr>
<tr>
<td>shlwapi</td>
<td>7</td>
</tr>
</tbody>
</table>

1674 summaries from 28 dll’s

3453 preconditions on arguments
2366 postconditions on return values
973 side effects on arguments
CodeHawk Binary Analyzer: Analysis

- .dll
- .exe
- data blocks
- function names
- function entry points
- disassemble and construct functions
- translate to CHIF
- library function summaries
- function summaries (xml)
- function invariants (xml)
- analyze CHIF
- invariants
- indirect jump targets
Analysis: Generate Invariants

Over-approximation on variable values and relationships at each location in the program that hold on all program behaviors

Intervals:
- 0x401a22: Eax = [ 0 .. 52 ]
- 0x401a28: Eax = [ 0 .. 53 ]
- 0x401a2a: Eax = [ 0 .. oo ]

Linear equalities:
- 0x401a28: Esi = Ebx + 4 * Ecx
- Edi = Edx + 4 * Ecx

Value sets:
- 0x40a102: Esi = Ebx_in + [ 0 .. 12 ] or
- Esi = Edx_in + [ 24 .. 36 ]
Value Sets

Disjunctive domain

- Developed by Balakrishnan, Reps
- Expressed by a list of (variable, interval) pairs
- Represents a disjunction of base-pointer plus range offset
- Provides a cheap weakly relational alternative to polyhedra
- Complexity is bounded by finite set of base pointers

CodeHawk adaptation:

- set of base-pointers not predetermined
- aggressive treatment of inconsistent conditions
Constraining Semantics

Addition: $x = y + z$

Legal only if at most one of $y,z$ is a pointer variable

<table>
<thead>
<tr>
<th></th>
<th>none</th>
<th>single</th>
<th>mult</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>none</td>
<td>single</td>
<td>mult</td>
<td>T</td>
</tr>
<tr>
<td>single</td>
<td>single</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>mult</td>
<td>mult</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Reps, Balakrishnan: Over-approximating semantics

Over-approximation of all behaviors

CodeHawk: Constraining semantics

Over-approximation of all legal behaviors
Iterate Analysis and Translation

- Each analysis run is performed bottom-up in the call graph
- Iterate until no new invariants are generated
- Reset invariants when
  - indirect jumps are resolved
  - function call side effects are identified
- Typically 10-20 iterations until convergence
### Iterate Analysis and Translation: Example

<table>
<thead>
<tr>
<th>run</th>
<th>fns anl.</th>
<th>stackp. (%)</th>
<th>reads (%)</th>
<th>writes (%)</th>
<th>coverage (%)</th>
<th>time (sec)</th>
<th>total time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>81</td>
<td>48.2</td>
<td>63.1</td>
<td>47.7</td>
<td>65.3</td>
<td>6.0</td>
<td>6.0</td>
</tr>
<tr>
<td>2</td>
<td>81</td>
<td>66.0</td>
<td>72.3</td>
<td>67.6</td>
<td>65.3</td>
<td>8.0</td>
<td>14.0</td>
</tr>
<tr>
<td>3</td>
<td>81</td>
<td>71.3</td>
<td>80.4</td>
<td>73.8</td>
<td>65.3</td>
<td>8.3</td>
<td>22.4</td>
</tr>
<tr>
<td>4</td>
<td>59</td>
<td>78.9</td>
<td>85.9</td>
<td>78.9</td>
<td>98.1</td>
<td>15.2</td>
<td>37.6</td>
</tr>
<tr>
<td>5</td>
<td>26</td>
<td>80.2</td>
<td>86.3</td>
<td>80.4</td>
<td>98.1</td>
<td>14.8</td>
<td>52.4</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td>80.2</td>
<td>86.3</td>
<td>80.4</td>
<td>98.1</td>
<td>14.5</td>
<td>66.9</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>80.3</td>
<td>86.3</td>
<td>80.4</td>
<td>98.7</td>
<td>13.6</td>
<td>80.5</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>80.3</td>
<td>86.3</td>
<td>80.4</td>
<td>98.7</td>
<td>10.2</td>
<td>90.7</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>80.3</td>
<td>86.3</td>
<td>80.4</td>
<td>98.7</td>
<td>9.6</td>
<td>100.3</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>80.3</td>
<td>86.3</td>
<td>80.4</td>
<td>98.7</td>
<td>7.0</td>
<td>107.2</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>80.3</td>
<td>86.3</td>
<td>80.4</td>
<td>98.7</td>
<td>1.1</td>
<td>108.3</td>
</tr>
</tbody>
</table>
Test and Evaluation: Corpus of executables

- Close to 700 executables and dll’s, including
  - putty.exe (500 KB)
  - nginx.exe (2.6MB)
  - openssl.exe (1.6MB)
  - jvm.dll (3.4MB)
  - java native libraries
  - ........
- Up to 8 MB in size (more than 18,000 functions)
- Originating from both C and C++ (and a few from Delphi)
- Automatic export of dll function summaries, imported in other dll’s
- Scripts to run analyses in parallel, respecting dll dependencies
Scalability: Analysis Times

Analysis times for all executables in the corpus
Quality Assessment: Measurements

Disassembly:

• instructions not recognized
• function coverage: % instructions included in functions
• function overlap: % instructions in two or more functions

Analysis:

• stack pointer precision: % known
• memory reads/writes
  • % known location
  • % known region
  • % unknown
• indirect jumps: % resolved
• indirect calls: % resolved
Precision: Indirect memory writes resolved

Percentage of indirect memory writes resolved for corpus of executables

- Instructions
- Percent writes resolved

Write precision

Instructions

1GB

100KB

10KB

1KB
Precision:
Improvements in Indirect Memory Writes Resolved

Improvement in percent indirect writes resolved for all executables

Aug 2015
May 2015
Reverse Engineering: what does the program do?
- information extraction
- API discovery
- establish relation with source code
- ........

Vulnerability Research:
- where are the vulnerabilities?
- how to get to the vulnerabilities?

Malware Analysis:
- what did/will/can the malware do?
- external inputs (from network, filesystem, registry, ....)
- external effects (outputs to network, filesystem, registry, .....)
- host/network-based indicators to aid in forensics and mitigation
CodeHawk Binary Analyzer: Graphical User Interface
CodeHawk Binary Analyzer: Graphical User Interface
CodeHawk Binary Analyzer: Graphical User Interface
Vulnerability Research: Memory Safety

**CHALLENGE:** What is memory safety of executables?

C

Memory safety can be mathematically defined based on the C language semantics as defined in the C standard.

X86

x86 instruction semantics provide very few constraints; memory safety conditions must be inferred from original C/C++ program.
C-Source Code Verification

```c
unsigned int seedify(char** argv) {
    char key[25];

    short x = 1;
    int d[3];
    d[x * x + 2 - 2] = 1;

    strcpy(key, argv[d[1]]);

    int i;
    unsigned int seed = 0;
    for (i = 0; i < KEY_LENGTH; i++) {
        seed += (unsigned int)(key[i]) << (i % 32);
    }

    return seed;
}
```

We can generate memory safety proof obligations based on the types provided and the C language semantics.

\[ \text{length(argv[d[1]] \leq 25) } \]
Binary Executable Verification

`strcpy(dest:&var.0047, src: arg.0004[4])`

We have to infer the size of the `strcpy` destination buffer.
<table>
<thead>
<tr>
<th>Address</th>
<th>Local</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4014fb</td>
<td>-76</td>
<td>save ebp</td>
</tr>
<tr>
<td>0x4014fc</td>
<td>-72</td>
<td>ebp := esp = (esp_in - 4)</td>
</tr>
<tr>
<td>0x4014fe</td>
<td>-56</td>
<td>save ebx</td>
</tr>
<tr>
<td>0x4014ff</td>
<td>-47</td>
<td>esp := esp - 68 = (esp_in - 76)</td>
</tr>
<tr>
<td>0x401502</td>
<td>-22</td>
<td>var.0022 := 1</td>
</tr>
<tr>
<td>0x401508</td>
<td>-20</td>
<td>edx := var.0022 = 1</td>
</tr>
<tr>
<td>0x40150c</td>
<td>-16</td>
<td>eax := var.0022 = 1</td>
</tr>
<tr>
<td>0x401510</td>
<td>-8</td>
<td>eax := eax * edx = 1</td>
</tr>
<tr>
<td>0x401513</td>
<td>-4</td>
<td>var.0056 := 1</td>
</tr>
<tr>
<td>0x40151b</td>
<td>4</td>
<td>nop</td>
</tr>
<tr>
<td>0x40151c</td>
<td>1</td>
<td>eax := var.0056 = 1</td>
</tr>
<tr>
<td>0x40151f</td>
<td>1</td>
<td>eax := eax * 4 = 4</td>
</tr>
<tr>
<td>0x401522</td>
<td>1</td>
<td>eax := eax + argv$1 = (4 + argv$1)</td>
</tr>
<tr>
<td>0x401525</td>
<td>1</td>
<td>eax := argv$1[4]</td>
</tr>
<tr>
<td>0x401527</td>
<td>1</td>
<td>[strcpy : src = eax ]</td>
</tr>
<tr>
<td>0x40152b</td>
<td>-76</td>
<td>eax := (ebp - 43) = (esp_in - 47)</td>
</tr>
<tr>
<td>0x40152e</td>
<td>-72</td>
<td>[strcpy : dest = eax ]</td>
</tr>
<tr>
<td>0x401531</td>
<td>-56</td>
<td>strcpy(dest :&amp;var.0047, src:(arg.0004)</td>
</tr>
<tr>
<td>0x401536</td>
<td>-47</td>
<td>var.0020 := 0</td>
</tr>
<tr>
<td>0x40153d</td>
<td>-47</td>
<td>var.0016 := 0</td>
</tr>
<tr>
<td>0x401544</td>
<td>-47</td>
<td>goto 0x401570</td>
</tr>
<tr>
<td>0x401546</td>
<td>-47</td>
<td>eax := (ebp - 43) = (esp_in - 47)</td>
</tr>
</tbody>
</table>
Vulnerability Research: Where does the input come from?

<table>
<thead>
<tr>
<th>Offset</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>+4</td>
<td>arg.0004</td>
</tr>
<tr>
<td>esp0</td>
<td>return-address</td>
</tr>
<tr>
<td>-4</td>
<td>saved ebp</td>
</tr>
<tr>
<td>-8</td>
<td>saved ebx</td>
</tr>
<tr>
<td>-22</td>
<td>var.0022 (1)</td>
</tr>
</tbody>
</table>

```
seedify (arg1:char **)
```

```
0x401531 strcpy(esp0 - 47, (arg.0004)[4])
```
main (argc:int, argv: char **)

init_deck(arg_1:(-440 + esp_in_1), arg_2:arg.0008)

init_deck(arg1:?, arg2: char **)

seedify(arg_1:arg.0008)

argv[1]

seedify(arg1: char **)

strcpy(esp0–47, (arg.0004)[4])

0x401ae2  init_deck(arg_1:(-440 + esp_in_1), arg_2:arg.0008)

0x4015ac  seedify(arg_1:arg.0008)

main(argc:int, argv: char **)

argv[1]

seedify(arg1: char **)

strcpy(esp0–47, (arg.0004)[4])

0x401531  strcpy(esp0–47, (arg.0004)[4])
Data propagation: A Look at the Call Graph
Interprocedural Data Propagation
Vulnerability Research: Quick Queries

all calls to sprintf with a stack buffer as destination
Vulnerability Research: Quick Queries

or all calls to strcpy with a heap buffer as destination

```
0x40a8b4  strcpy(dest:safemalloc_rtn_0x40a8ab, src:arg.0004)
0x40a8ff  strcpy(dest:safemalloc_rtn_0x40a8f4, src:arg.0004)
0x40b2f1  strcpy(dest:(32 + safemalloc_rtn_0x40b2cb), src:arg.0004)
0x40be7b  strcpy(dest:(8 + safemalloc_rtn_0x40be69), src:var.0540)
0x4448fa  strcpy(dest:safemalloc_rtn_0x4448e8, src:(-8236 + esp_in))
0x4474c8  strcpy(dest:safemalloc_rtn_0x4474b0, src:arg.0004)
0x4510f2  strcpy(dest:malloc_rtn_0x4510e4, src:var.0020)
0x4537a2  strcpy(dest:malloc_rtn_0x453791, src:var.0008)
0x4540bb  strcpy(dest:malloc_rtn_0x4540ae, src:arg.0004)
```
Malware Analysis

- Collect corpus of malware
- Collect information on
  - what data is retrieved from the computer?
  - what data is received from the network?
  - what data is sent to the network?
  - what actions are performed on the computer/peripherals?
- Collect
  - host-based indicators (filenames, registry keys, environment variables, ...)
  - network-based indicators (ip addresses, domain names, ....)
  - input indicators (strings compared against)
  - output indicators (format strings)
- Detect suspicious activity
Conclusions

- Deep semantic analysis of x86 executables
- Continuous improvement in scalability, precision, and robustness driven by automatic test and evaluation
- Contacts with several companies interested in
  - vulnerability research
  - malware analysis
- Commercialization is a slow and difficult process
Conclusions

Static Analysis

THEORY → PRACTICE

requires a LOT of

• expertise
• experience
• experimentation
• engineering
Conclusion

in theory

practice and theory are the same

in practice

they are not

Call for a new engineering discipline: Static Analysis Engineering