



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

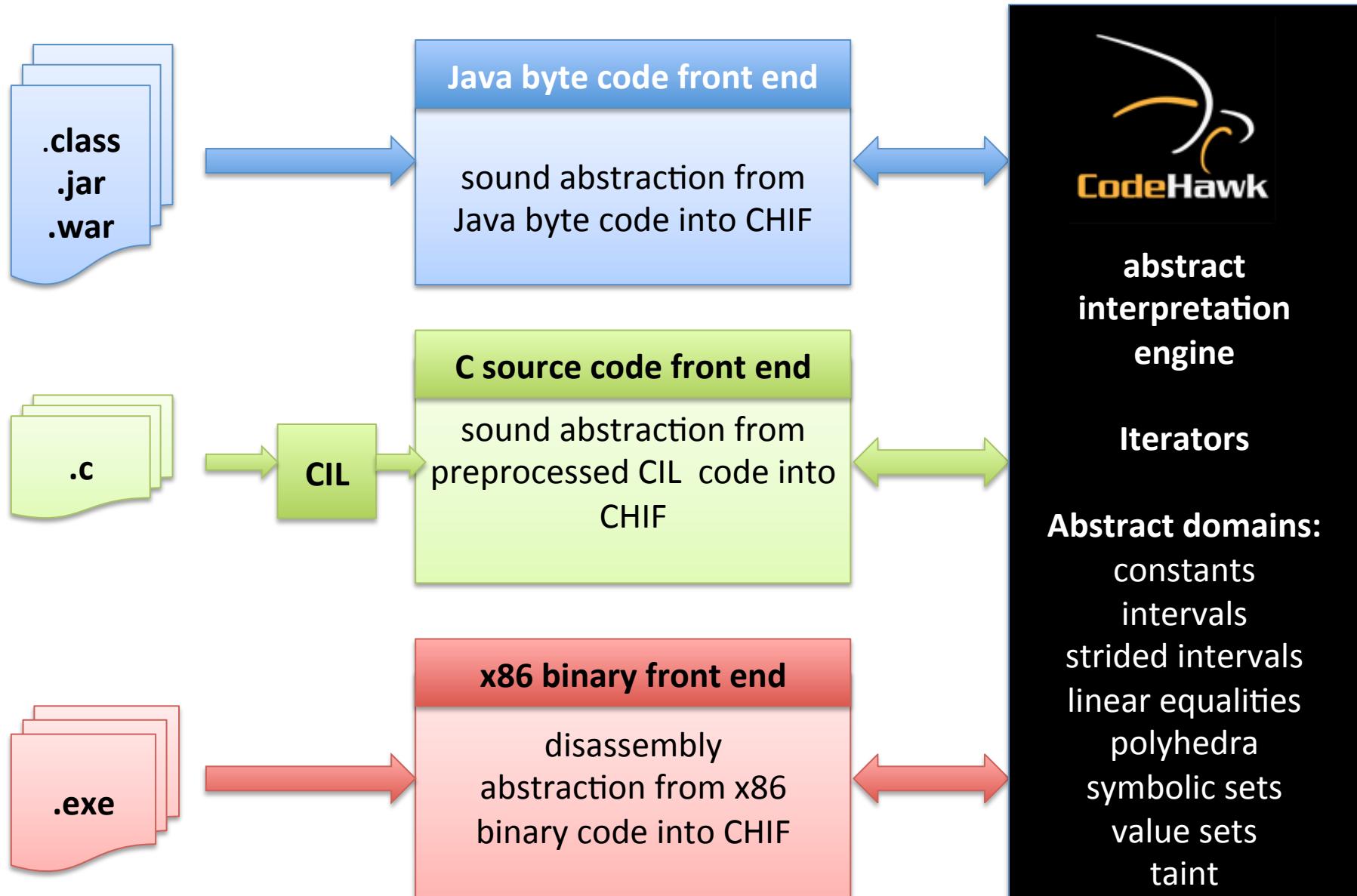
Location: Palo Alto, California

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





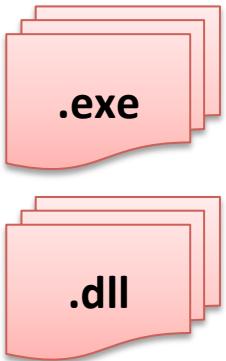
Outline



- **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



x86 front end



abstract
interpretation
engine

Iterators

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

Targeted at

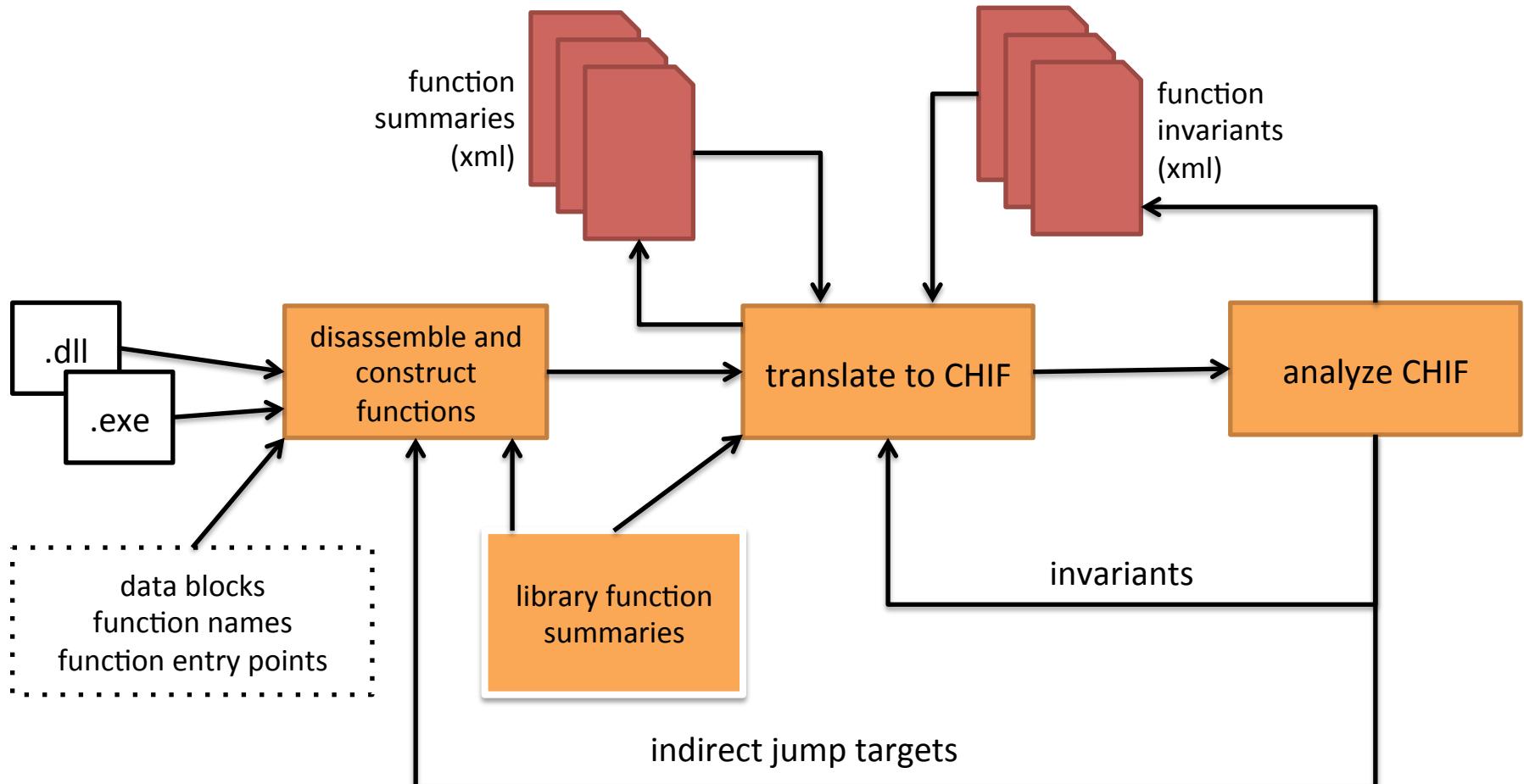
- vulnerability researchers
- reverse engineers
- software assurance centers
- malware analysis/forensics
-

Complementary to

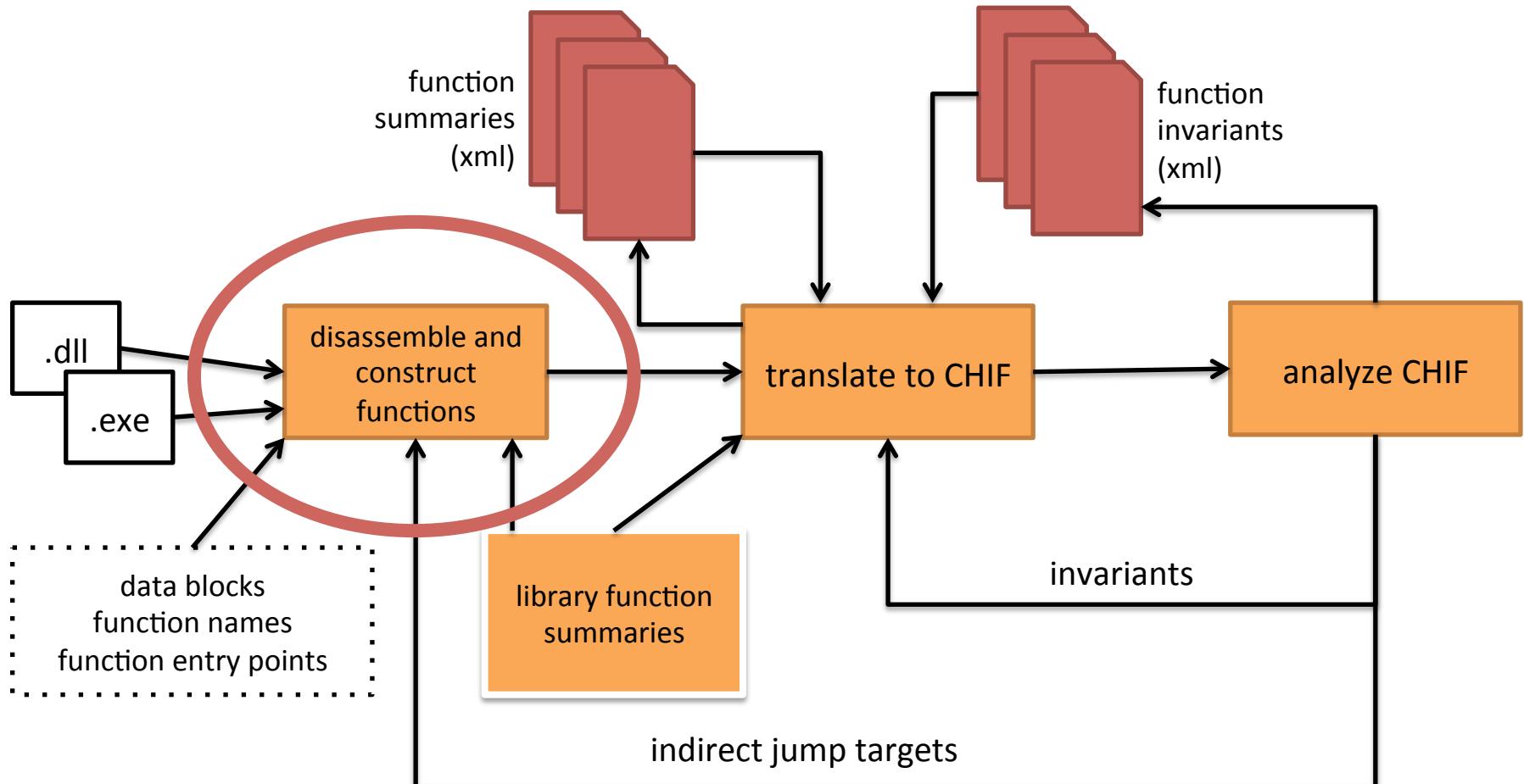
- IDA Pro
- objdump
-

Goal: produce a commercial product

CodeHawk Binary Analyzer: Architecture

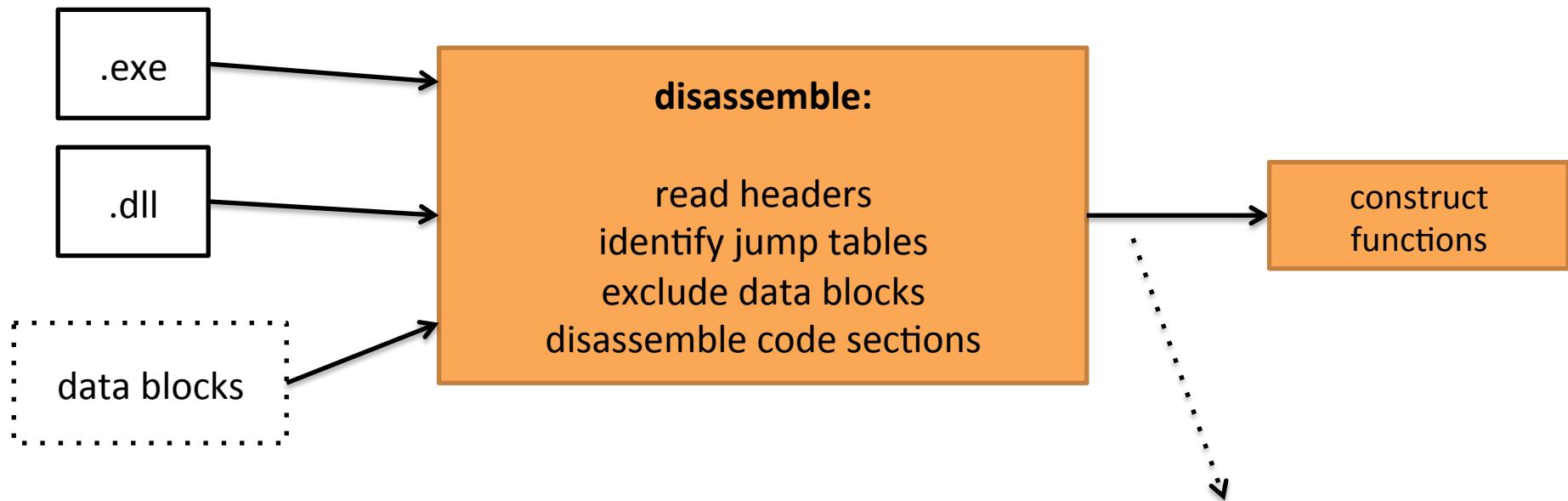


CodeHawk Binary Analyzer: Disassembler



CodeHawk Binary Analyzer: Disassembler

32-bit PE



Disassembly method: linear sweep

Recognizes ~900 opcodes (out of ~1700),
including SSE and AVX instructions

~ 160 internal instruction types:

Add

Mov

Push

Pop

Jcc

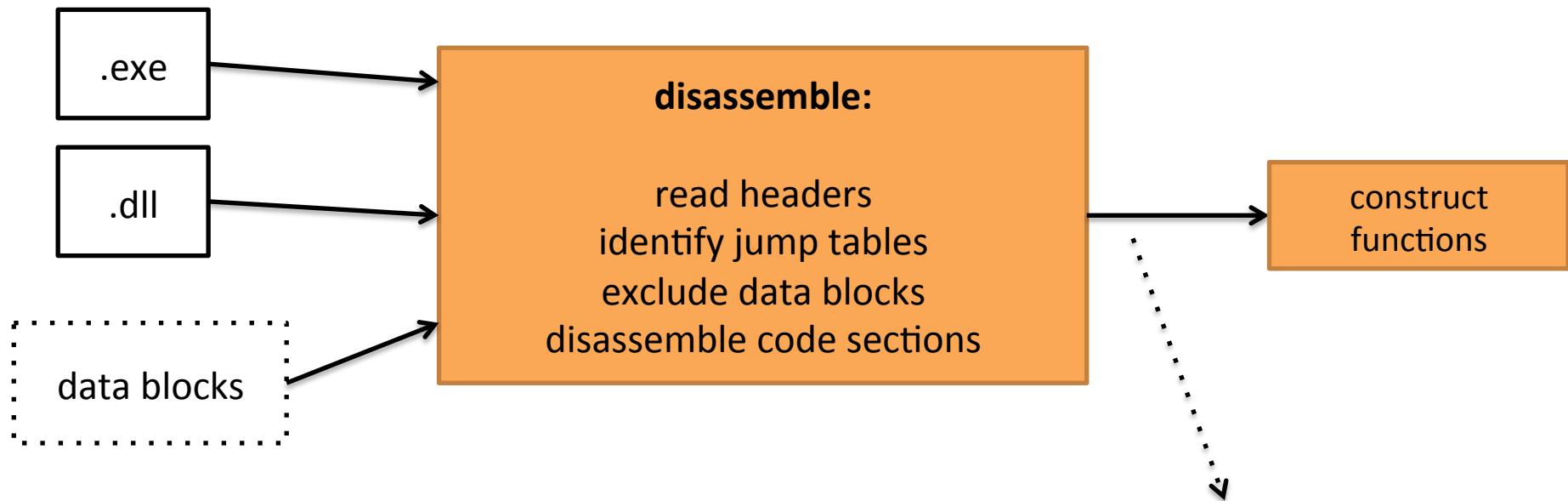
...

Fragment from nginx.exe

B 0x570fe9	66 0f ef c0	pxor %xmm0, %xmm0
0x570fed	51	push ecx
0x570fee	53	push ebx
0x570fef	8b c1	mov eax, ecx
0x570ff1	83 e0 0f	and eax, \$0xf
0x570ff4	85 c0	test eax, eax
0x570ff6	75 7f	jnz 0x571077
B 0x570ff8	8b c2	mov eax, edx
0x570ffa	83 e2 7f	and edx, \$0x7f
0x570ffd	c1 e8 07	shr eax, \$0x7
0x571000	74 37	jz 0x571039
B 0x571002	8d a4 24 00 00 00 00	lea esp, 0x0(%esp,,1)
B 0x571009	66 0f 7f 01	movdqa (%ecx), %xmm0
0x57100d	66 0f 7f 41 10	movdqa 0x10(%ecx), %xmm0
0x571012	66 0f 7f 41 20	movdqa 0x20(%ecx), %xmm0
0x571017	66 0f 7f 41 30	movdqa 0x30(%ecx), %xmm0
0x57101c	66 0f 7f 41 40	movdqa 0x40(%ecx), %xmm0
0x571021	66 0f 7f 41 50	movdqa 0x50(%ecx), %xmm0
0x571026	66 0f 7f 41 60	movdqa 0x60(%ecx), %xmm0
0x57102b	66 0f 7f 41 70	movdqa 0x70(%ecx), %xmm0
0x571030	8d 89 80 00 00 00	lea ecx, 0x80(%ecx)
0x571036	48	dec eax
0x571037	75 d0	jnz 0x571009
B 0x571039	85 d2	test edx, edx
0x57103b	74 37	jz 0x571074

CodeHawk Binary Analyzer: Disassembler

32-bit PE



Disassembly method: linear sweep

Recognizes ~900 opcodes (out of ~1700),
including SSE and AVX instructions

~ 160 internal instruction types:

Add

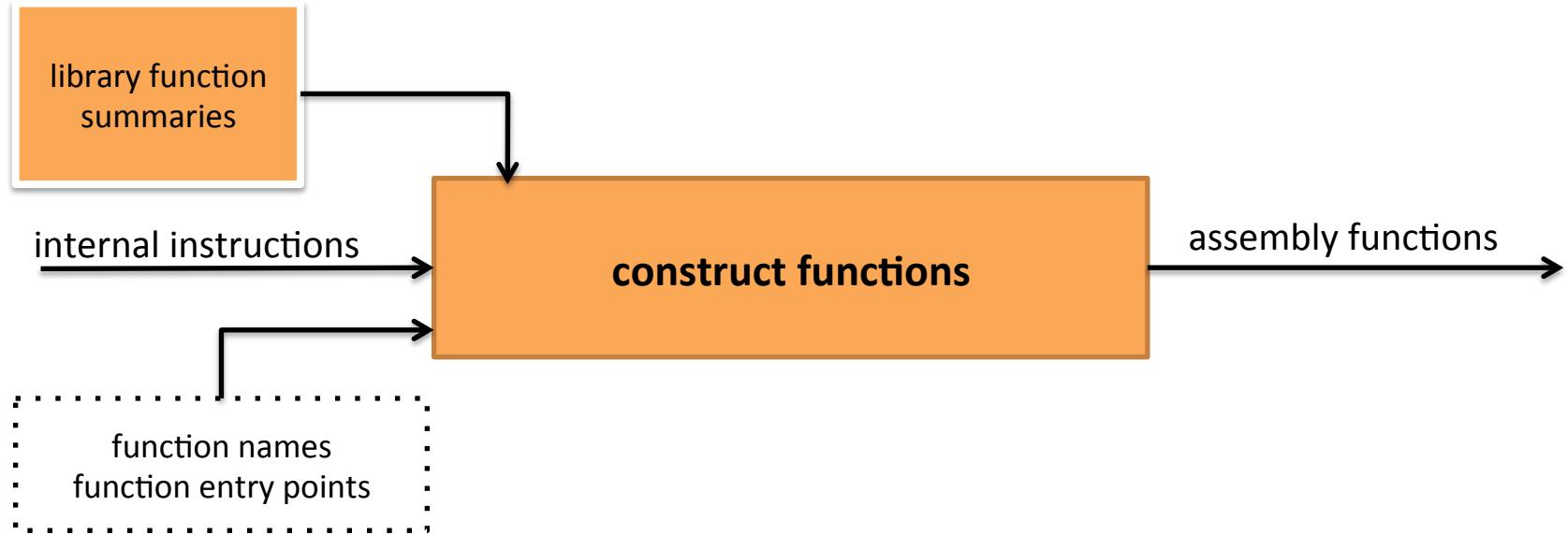
Mov

Push

Pop

Jcc

...



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

CcOverflow	CcNotOverflow	-> [OFlag]
CcCarry	CcNotCarry	-> [CFlag]
CcZero	CcNotZero	-> [ZFlag]
CcBelowEqual	CcAbove	-> [CFlag ; ZFlag]
CcSign	CcNotSign	-> [SFlag]
CcParityEven	CcParityOdd	-> [PFlag]
CcLess	CcGreaterEqual	-> [SFlag ; OFlag]
CcLessEqual	CcGreater	-> [ZFlag ; SFlag ; OFlag]

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
```

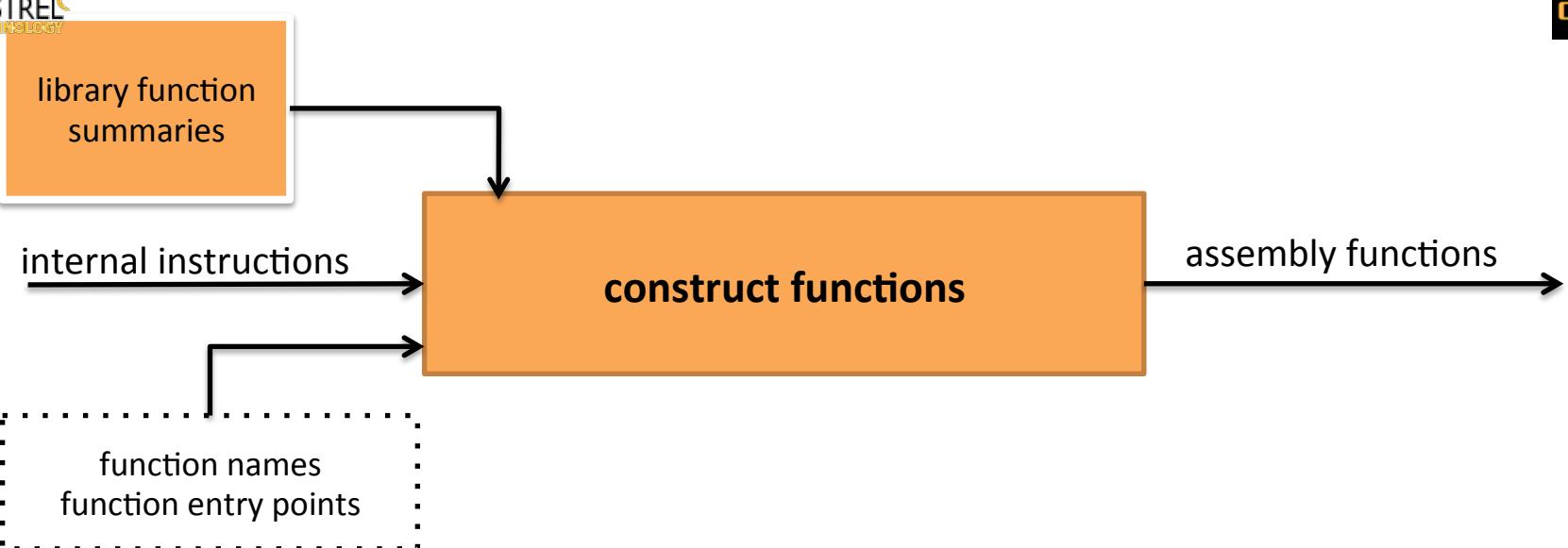
```
cmp eax, $0x8  
jnz 0x47e9ca  
cmp 0x8c(%edi), $0x1  
jnz 0x47e9ca  
mov edx, 0x4(%esi)  
push eax  
push $0x5fdd74  
push edx  
call 0x401253  
add esp, $0xc  
test eax, eax  
mov eax, $0x4  
jz 0x47e9cf  
mov eax, $0x14  
pop esi  
pop edi  
pop ebx  
ret
```

jump not zero

jump not zero

jump zero

CodeHawk Binary Analyzer: Construct Functions



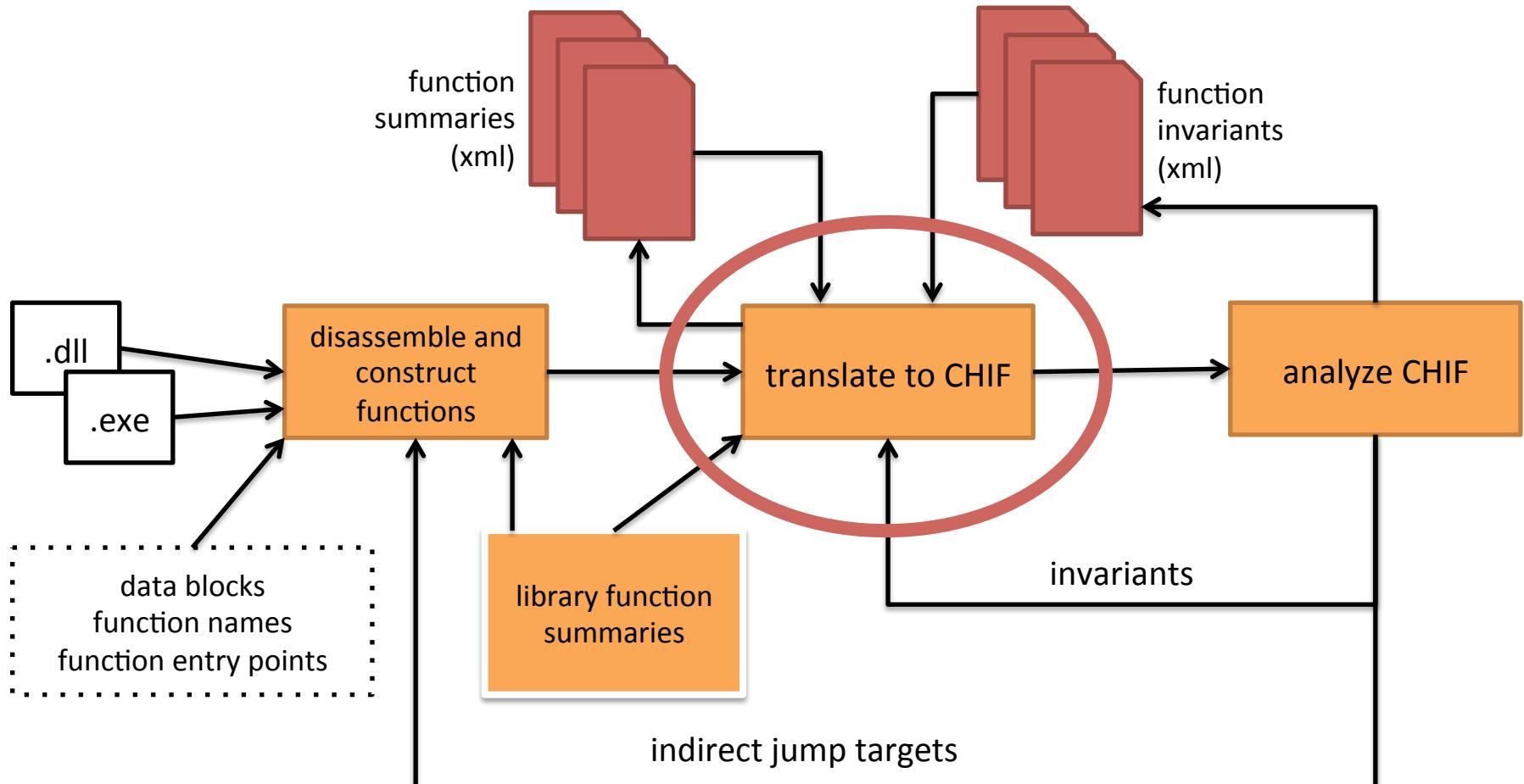
1. Collect direct call targets, combine with user-provided function entry points
2. For every function entry point:
 1. Identify non-returning function calls
 2. Identify basic blocks
 3. Construct control flow graph
 4. Connect conditional jumps with test expressions
 5. Identify function call arguments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

Connect function calls with their arguments (non-gcc)

```
B 0x402348 53  
0x402349 56  
0x40234a 8b 74 24 14  
0x40234e 8b 46 08  
0x402351 57  
0x402352 68 28 bf 5e 00  
0x402357 6a 00  
0x402359 50  
0x40235a 6a 06  
0x40235c e8 49 f2 ff ff  
0x402361 8b 46 04  
0x402364 8d 5e 38  
0x402367 68 e8 03 00 00  
0x40236c 50  
0x40236d c7 43 04 00 00 00 00  
0x402374 c7 43 08 64 00 00 00  
0x40237b c7 43 0c 0a 00 00 00  
0x402382 89 43 10  
0x402385 e8 9f f0 ff ff  
0x40238a 83 c4 1c  
0x40238d 89 03  
0x40238f 85 c0  
0x402391 75 08  
B 0x402393 5e  
0x402394 5b  
0x402395 83 c8 ff  
0x402398 5f  
0x402399 59  
0x40239a c3
```

```
push ebx  
push esi  
mov esi, 0x14(%esp,,1)  
mov eax, 0x8(%esi)  
push edi  
4 push $0x5ebf28  
3 push $0x0  
2 push eax  
1 push $0x6  
call 0x4015aa  
mov eax, 0x4(%esi)  
lea ebx, 0x38(%esi)  
2 push $0x3e8  
1 push eax  
mov 0x4(%ebx), $0x0  
mov 0x8(%ebx), $0x64  
mov 0xc(%ebx), $0xa  
mov 0x10(%ebx), eax  
call 0x401429  
add esp, $0x1c  
mov (%ebx), eax  
test eax, eax  
jnz 0x40239b  
pop esi  
pop ebx  
or eax, $-0x1  
pop edi  
pop ecx  
ret
```

CodeHawk Binary Analyzer: Abstraction





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

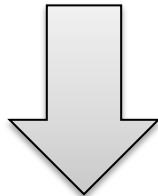
does not have:

- 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)

$op1 := op1 + op2$

Mov (op1,op2)

$op1 := op2$

Push op

$esp := esp - 4 ; mem[esp] := op$

Some involve limited non-determinism:

AddCarry (op1,op2)

$op1 := op1 + op2$ –or– $op1 := op1 + op2 + 1$

Some abstract completely:

PackedAlignRight (op, . . .)

$op := \text{TOP}$ (destination is abstracted)



Abstraction into CHIF



Some are complex:

RepMofs (width, dst, src)

```
(* -----
* RepMofs: 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

mov 0xc(eax), ecx

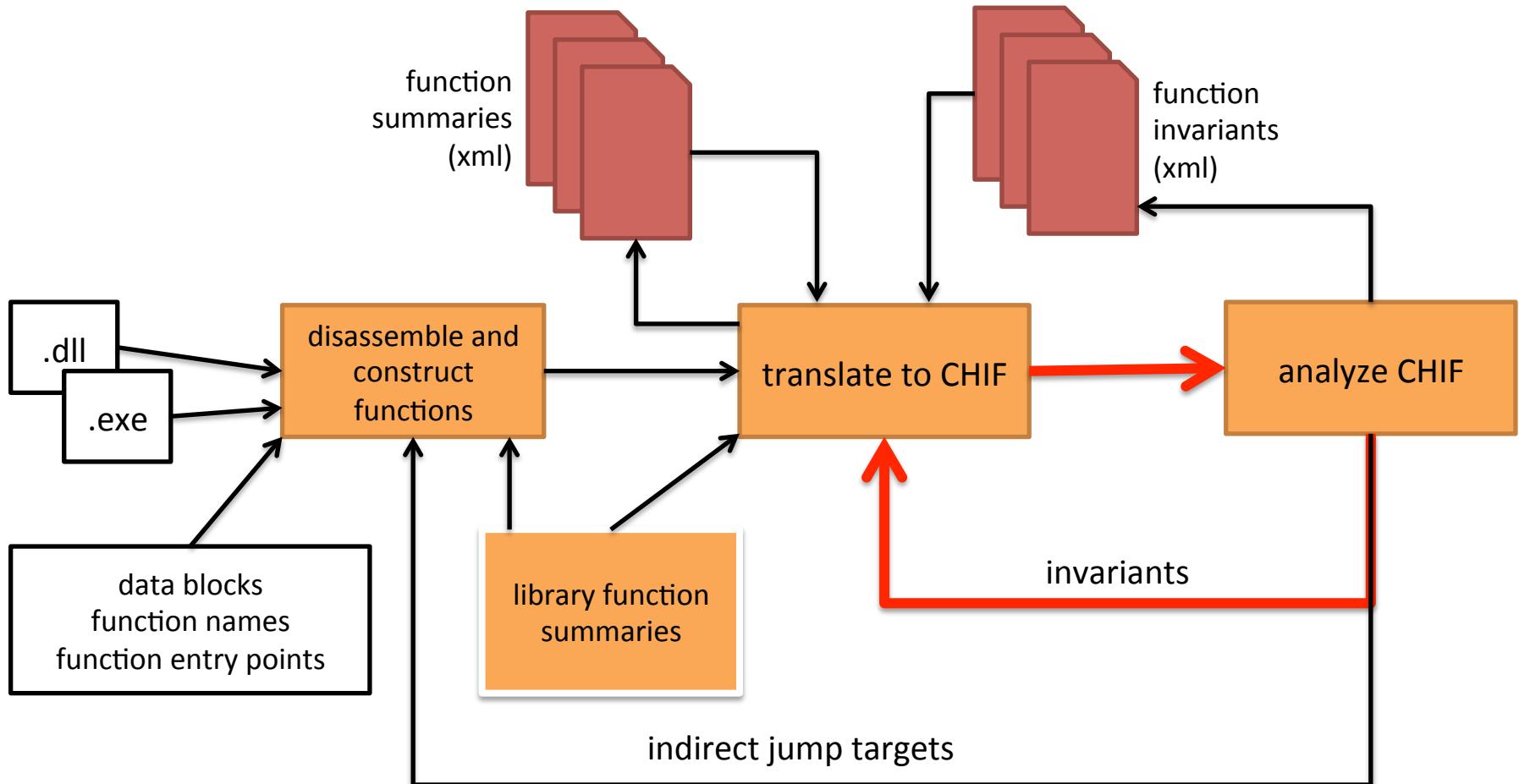
assign the contents of register ecx to the memory location
pointed to by the contents of register eax plus twelve

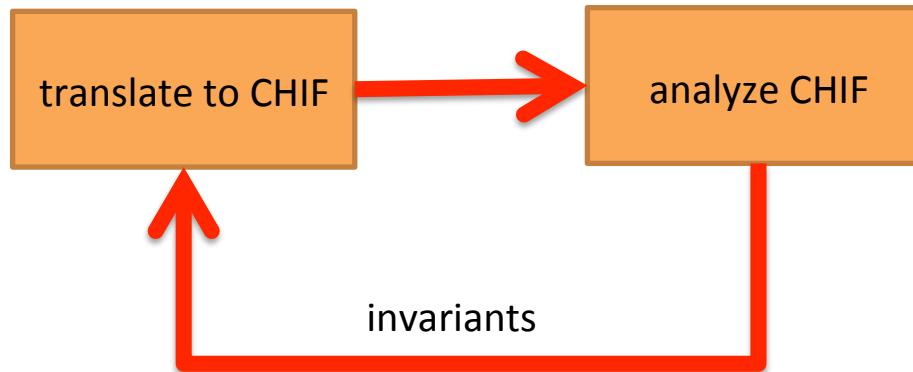
We have to know the value of eax before we can abstract this instruction

APPROACH

Incremental creation of variables by iterative analysis

Abstraction into CHIF

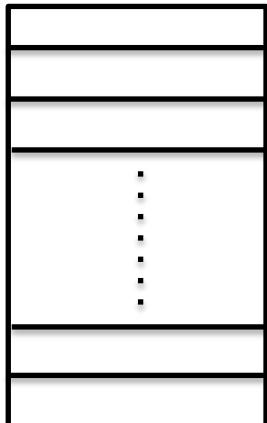




Start with

Eax Ebx Ecx Edx Esp Ebp Esi Edi

generic memory location: L

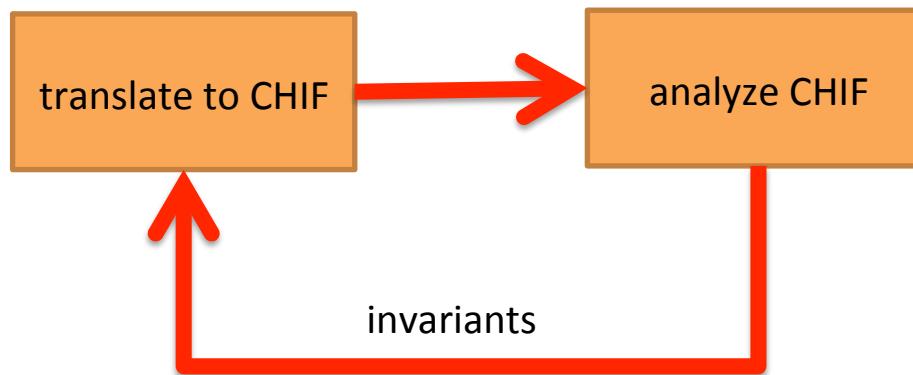


global data

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

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

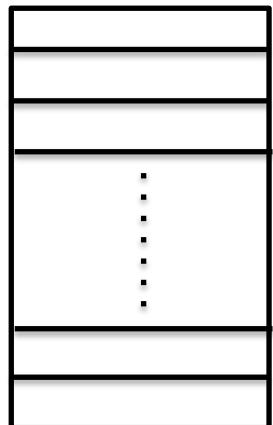
and similar for the other 150 instruction types



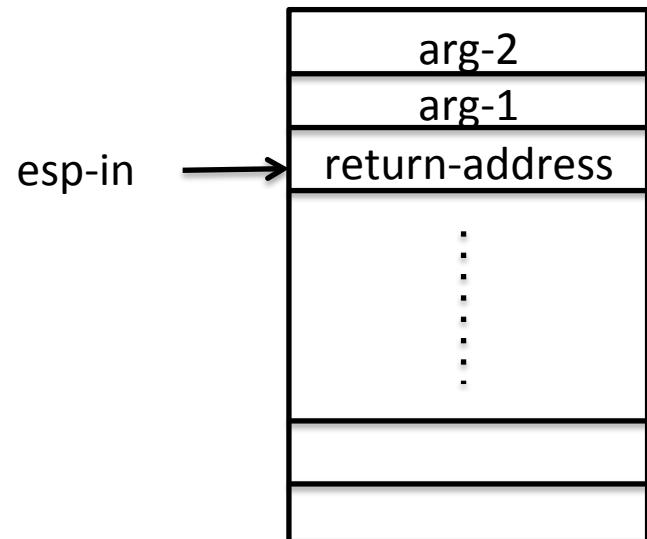
... and incrementally work towards (for each function)



generic memory location: L



global data



esp-in



local stack frame

Analysis: Resolve Memory References

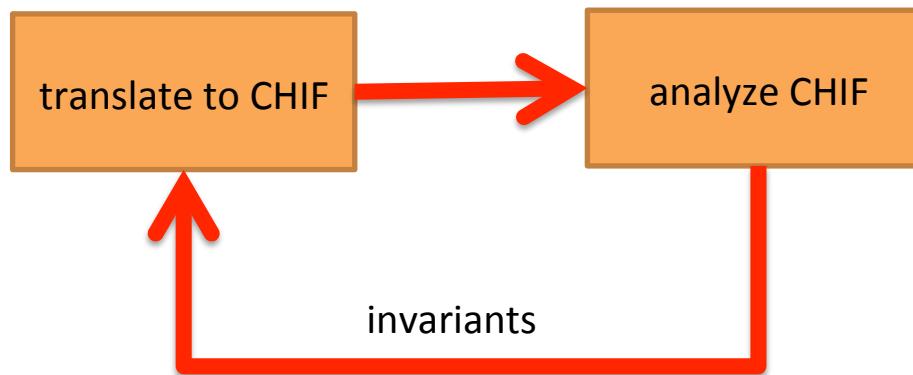
Initially: $Esp = Esp_in$

with Esp_in the address of the return-address

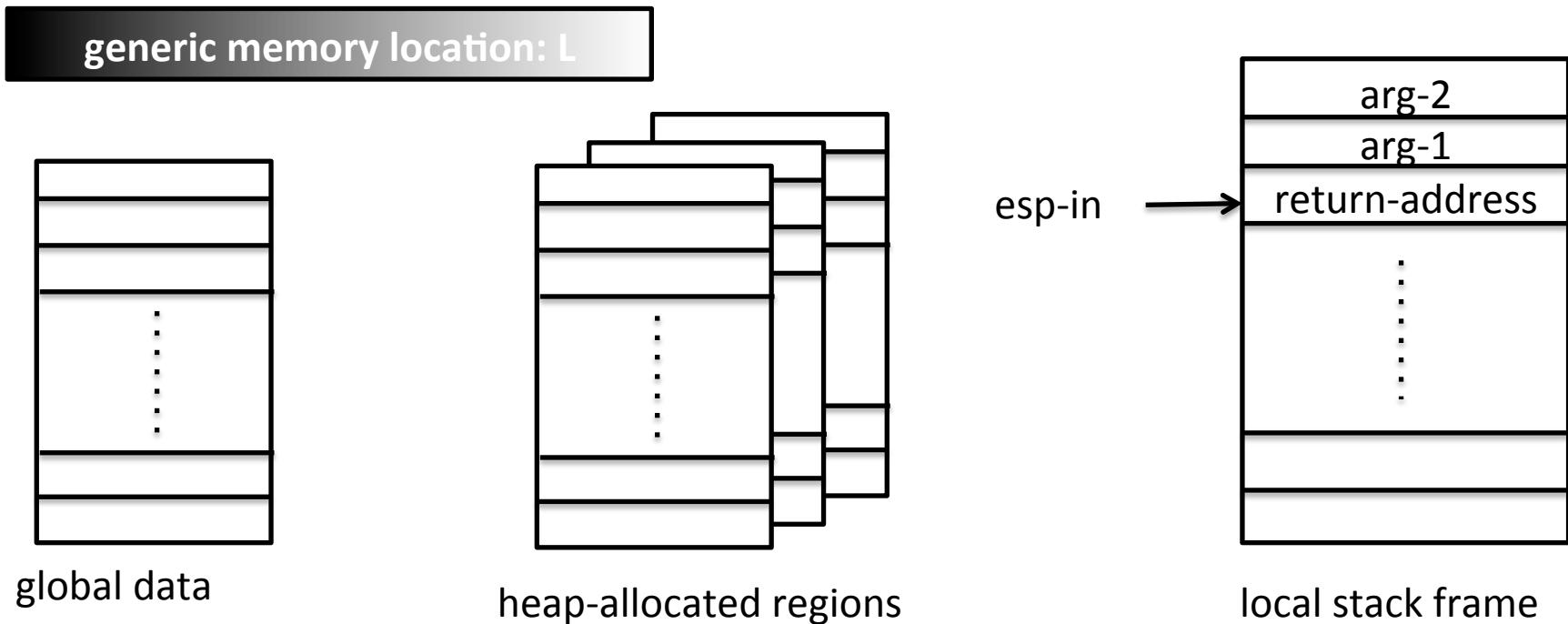
with invariant $Eax = Esp_in - 4$

`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)



function seedify

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

		invariants	2nd translation
0x4014fb	push ebp	esp = esp0	esp := esp - 4 ; var.0004 := ebp
0x4014fc	mov ebp, esp	esp = esp0 - 4	ebp := esp
0x4014fe	push ebx	ebp = esp0 - 4	esp := esp - 4 ; var.0008 := ebx
0x4014ff	sub esp, \$0x44	esp = esp0 - 8	esp := esp - 68
0x401502	mov -0x12(ebp), \$0x1	ebp = esp0 - 4	var.0022 := 1
0x401508	movsx edx, -0x12(ebp)	ebp = esp0 - 4	edx := var.0022
0x40150c	movsx eax, -0x12(ebp)	ebp = esp0 - 4	eax := var.0022
0x401510	imul eax, eax, edx		eax := eax * edx
0x401513	mov -0x38(ebp,eax,4), \$0x1	ebp = esp0 - 4	L? := 1
0x40151c	mov eax, -0x34(ebp)	ebp = esp0 - 4	eax := var.0056
0x40151f	shl eax, \$0x2		eax := eax * 4
0x401522	add eax, 0x8(ebp)	ebp = esp0 - 4	eax := eax + arg.0004
0x401525	mov eax, (eax)		eax := TOP
0x401527	mov 0x4(esp,,1), eax	esp = esp0 - 76	[strcpy:src := eax]
0x40152b	lea eax, -0x2b(ebp)	ebp = esp0 - 4	eax := esp0 - 47
0x40152e	mov 0x0(esp,,1), eax	esp = esp0 - 76	[strcpy:dst := eax]
0x401531	call 0x407020		call strcpy

		invariants	3rd translation
0x4014fb	push ebp	esp = esp0	esp := esp - 4 ; var.0004 := ebp
0x4014fc	mov ebp, esp	esp = esp0 - 4	ebp := esp
0x4014fe	push ebx	ebp = esp0 - 4	esp := esp - 4 ; var.0008 := ebx
0x4014ff	sub esp, \$0x44	esp = esp0 - 8	esp := esp - 68
0x401502	mov -0x12(ebp), \$0x1	ebp = esp0 - 4	var.0022 := 1
0x401508	movsx edx, -0x12(ebp)	ebp = esp0 - 4	edx := var.0022
0x40150c	movsx eax, -0x12(ebp)	ebp = esp0 - 4	eax := var.0022
0x401510	imul eax, eax, edx	eax = edx = 1	eax := eax * edx
0x401513	mov -0x38(ebp,eax,4), \$0x1	ebp = esp0 - 4, eax = 1	var.0056 := 1
0x40151c	mov eax, -0x34(ebp)	ebp = esp0 - 4	eax := var.0056
0x40151f	shl eax, \$0x2		eax := eax * 4
0x401522	add eax, 0x8(ebp)	ebp = esp0 - 4	eax := eax + arg.0004
0x401525	mov eax, (eax)		eax := TOP
0x401527	mov 0x4(esp,,1), eax	esp = esp0 - 76	[strcpy:src := eax]
0x40152b	lea eax, -0x2b(ebp)	ebp = esp0 - 4	eax := esp0 - 47
0x40152e	mov 0x0(esp,,1), eax	esp = esp0 - 76	[strcpy:dst := esp0-47]
0x401531	call 0x407020		call strcpy

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
ebp = esp0 - 4
eax = edx = 1
ebp = esp0 - 4, eax = 1
ebp = esp0 - 4
eax = 1
ebp = esp0 - 4
esp = esp0 - 76
ebp = esp0 - 4
esp = esp0 - 76

4th translation

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]]
eax := esp0 - 47
[strcpy:dst := esp0 - 47]
call strcpy

2nd translation

... ; var.0004 := ebp

ebp := esp

... ; var.0008 := ebx

esp := esp - 68

var.0022 := 1

edx := var.0022

eax := var.0022

eax := eax * edx

L? := 1

eax := var.0056

eax := eax * 4

eax := eax + arg.0004

eax := TOP

[strcpy:src := eax]

eax := esp0 - 47

[strcpy:dst := eax]

call strcpy

3rd translation

... ; var.0004 := ebp

ebp := esp

... ; 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 := eax + arg.0004

eax := TOP

[strcpy:src := eax]

eax := esp0 - 47

[strcpy:dst := esp0-47]

call strcpy

4th translation

... ; var.0004 := ebp

ebp := esp

... ; 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]]

eax := esp0 - 47

[strcpy:dst := esp0 - 47]

call strcpy

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	cmp eax, \$0x8
0x47e9a4	75 24	jnz 0x47e9ca
B 0x47e9a6	83 bf 8c 00 00 00 01	cmp 0x8c(%edi), \$0x1
0x47e9ad	75 1b	jnz 0x47e9ca
B 0x47e9af	8b 56 04	mov edx, 0x4(%esi)
0x47e9b2	50	push eax
0x47e9b3	68 74 dd 5f 00	push \$0x5fdd74
0x47e9b8	52	push edx
0x47e9b9	e8 95 28 f8 ff	call 0x401253
0x47e9be	83 c4 0c	add esp, \$0xc
0x47e9c1	85 c0	test eax, eax
0x47e9c3	b8 04 00 00 00	mov eax, \$0x4
0x47e9c8	74 05	jz 0x47e9cf
B 0x47e9ca	b8 14 00 00 00	mov eax, \$0x14
B 0x47e9cf	5e	pop esi
0x47e9d0	5f	pop edi
0x47e9d1	5b	pop ebx
0x47e9d2	c3	ret

Abstraction into CHIF: Conditional Jumps

(adapted from Balakrishnan, Reps)

Condition	Predicate
jc	$y <_u x$
jnc	$y \geq_u x$
jz	$x = y$
jnz	$x \neq y$
jbe	$y \leq_u x$
ja	$y >_u x$
jl	$y < x$
jge	$y \geq x$
jle	$y \leq x$
jg	$y > x$
js	$y < x$
jns	$y \geq x$

Condition	Predicate
jz	$x = 1$
jnz	$x \neq 1$
js	$x \leq 0$
jns	$x > 0$
jl	$x \leq 0$
jge	$x > 0$
jle	$x \leq 1$
jg	$x > 1$

Condition	Flags	Predicate
jz	ZF	$x = 0$
jnz	$\neg ZF$	$x \neq 0$
js	SF	$x < 0$
jns	$\neg SF$	$x \geq 0$
jbe	ZF	$x = 0$
ja	$\neg ZF$	$x \neq 0$
jl	SF	$x < 0$
jge	$\neg SF$	$x \geq 0$
jle	$ZF \vee SF$	$x \leq 0$
jg	$\neg ZF \wedge \neg SF$	$x > 0$

dec x

test x,x

cmp y,x

Fragment from nginx-1.2.7

```
0x47e9a1 83 f8 08          cmp eax, $0x8
0x47e9a4 75 24             jnz 0x47e9ca
0x47e9a6 83 bf 8c 00 00 00 01  cmp 0x8c(%edi), $0x1
0x47e9ad 75 1b             jnz 0x47e9ca
0x47e9af 8b 56 04           mov edx, 0x4(%esi)
0x47e9b2 50                 push eax
0x47e9b3 68 74 dd 5f 00       push $0x5fdd74
0x47e9b8 52                 push edx
0x47e9b9 e8 95 28 f8 ff      call 0x401253
0x47e9be 83 c4 0c           add esp, $0xc
0x47e9c1 85 c0               test eax, eax
0x47e9c3 b8 04 00 00 00       mov eax, $0x4
0x47e9c8 74 05              jz 0x47e9cf
```

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
```

→ 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

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

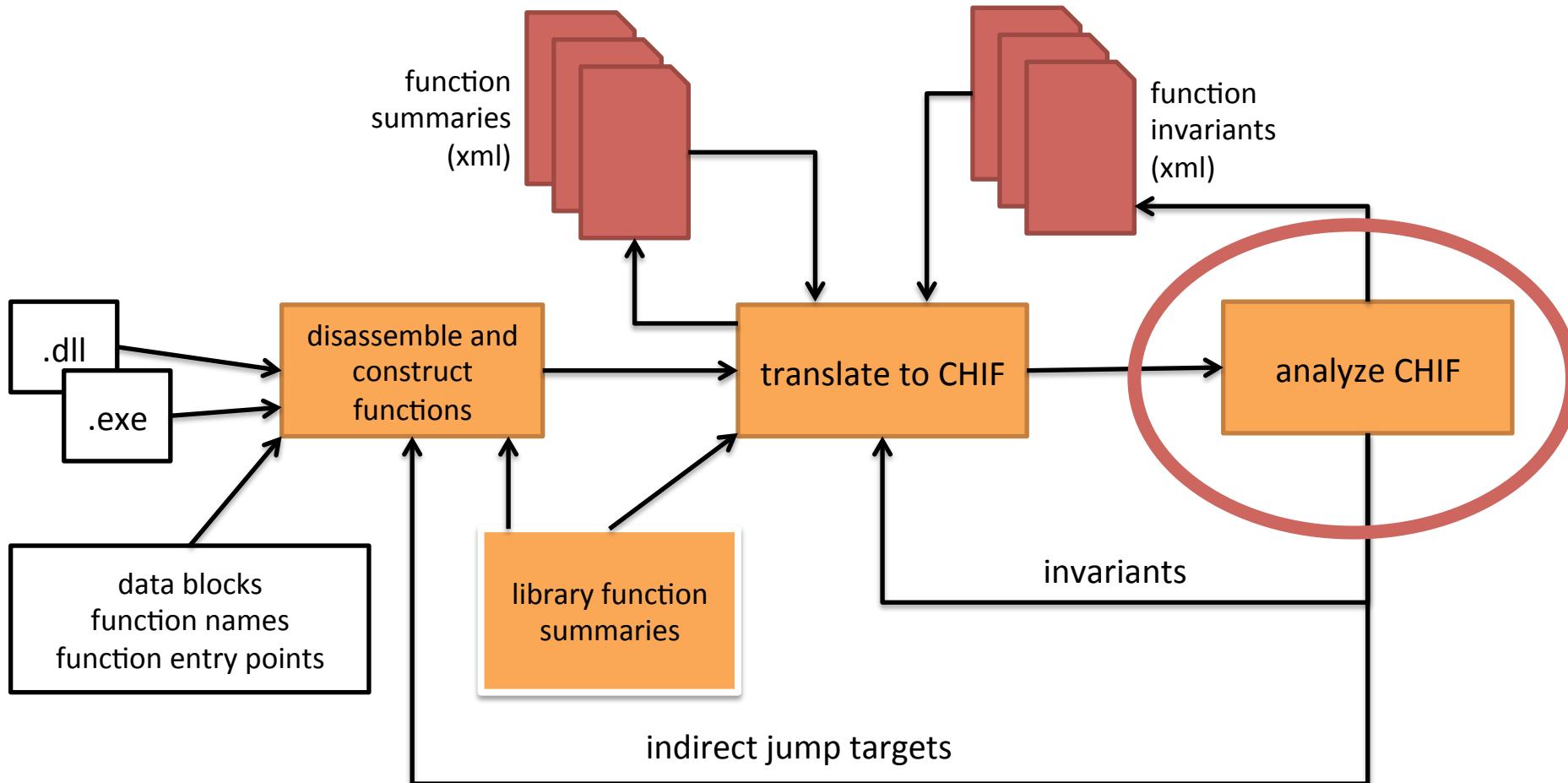
Library Function Summaries

dll	function summaries	dll	function summaries	dll	function summaries
advapi32	179	mswsock	2	spools	2
comctl32	9	netapi32	3	urlmon	2
comdlg32	28	ole32	29	user32	382
crypt32	3	oleaut32	14	version	10
dwmapi	5	opengl32	4	wininet	53
gdi32	181	psapi	2	winmm	32
imm32	21	secur32	5	winspool	21
kernel32	518	shell32	20	ws2_32	60
msvcrt	147	shlwapi	7	wsock32	26
msvfw32	6				

1674 summaries from 28 dll's

3453 preconditions on arguments
 2366 postconditions on return values
 973 side effects on arguments

CodeHawk Binary Analyzer: Analysis





Analysis: Generate Invariants



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

0x401a22: $Eax = [0 .. 52]$

intervals: 0x401a28: $Eax = [0 .. 53]$

0x401a2a: $Eax = [0 .. oo]$

$Ebp = Esp_in - 4$

linear equalities: 0x401a28: $Esi = Ebx + 4 * ECX$

$Edi = Edx + 4 * ECX$

$Esi = Ebx_in + [0 .. 12]$

value sets: 0x40a102: 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

Reps, Balakrishnan:
Over-approximating semantics

Over-approximation of *all* behaviors

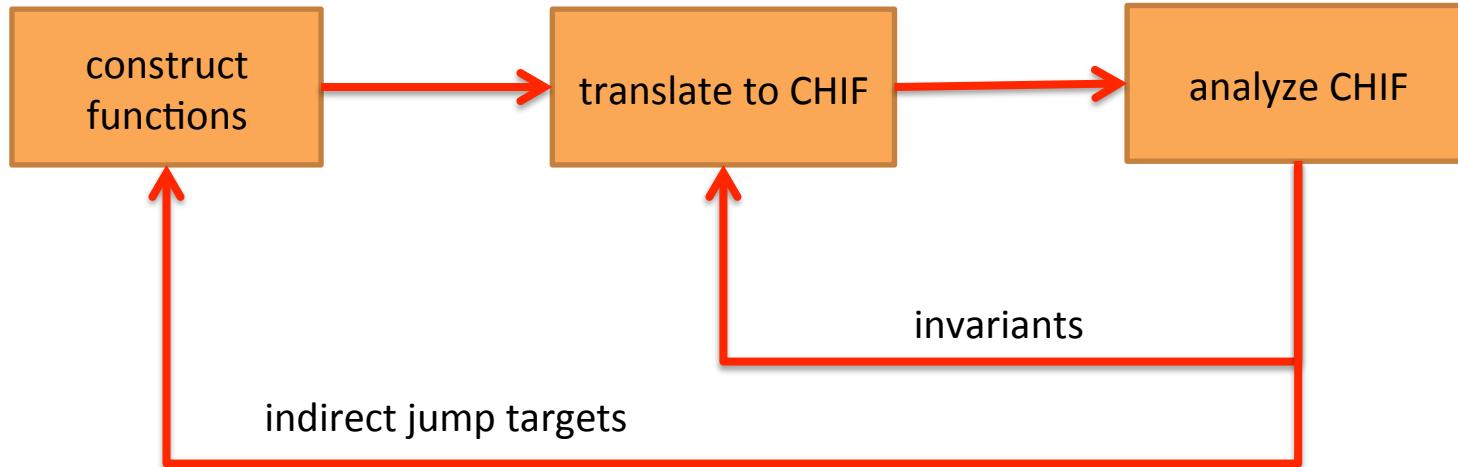
+	none	single	mult	T
none	none	single	mult	T
single	single	T	T	T
mult	mult	T	T	T
T	T	T	T	T

CodeHawk:
Constraining semantics

Over-approximation of *all legal* behaviors

+	none	single	mult	T
none	none	single	mult	T
single	single	bot	bot	single
mult	mult	bot	bot	mult
T	T	single	mult	T

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



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

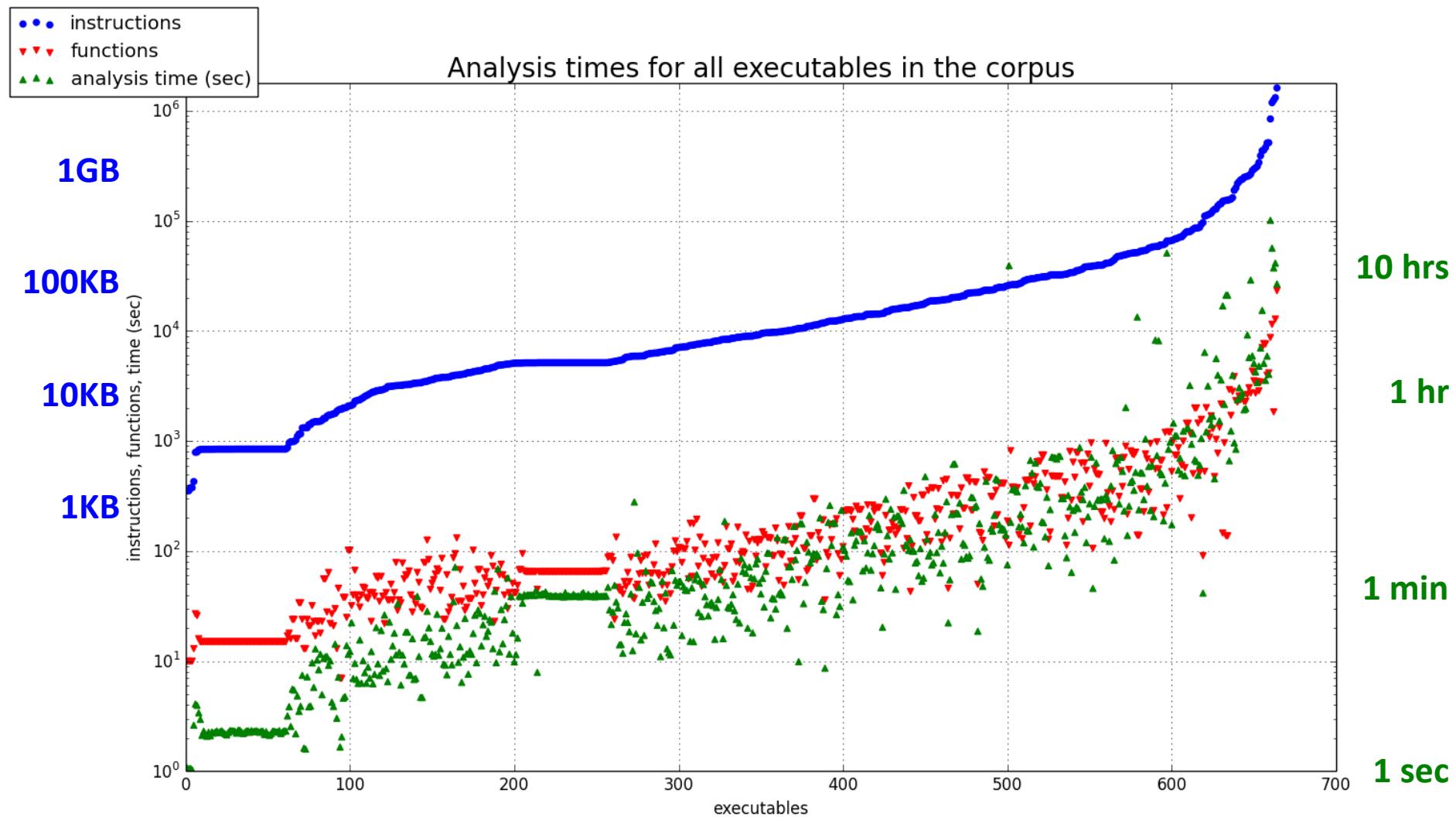


Test and Evaluation: Corpus of executables



- 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





Quality Assessment: Measurements



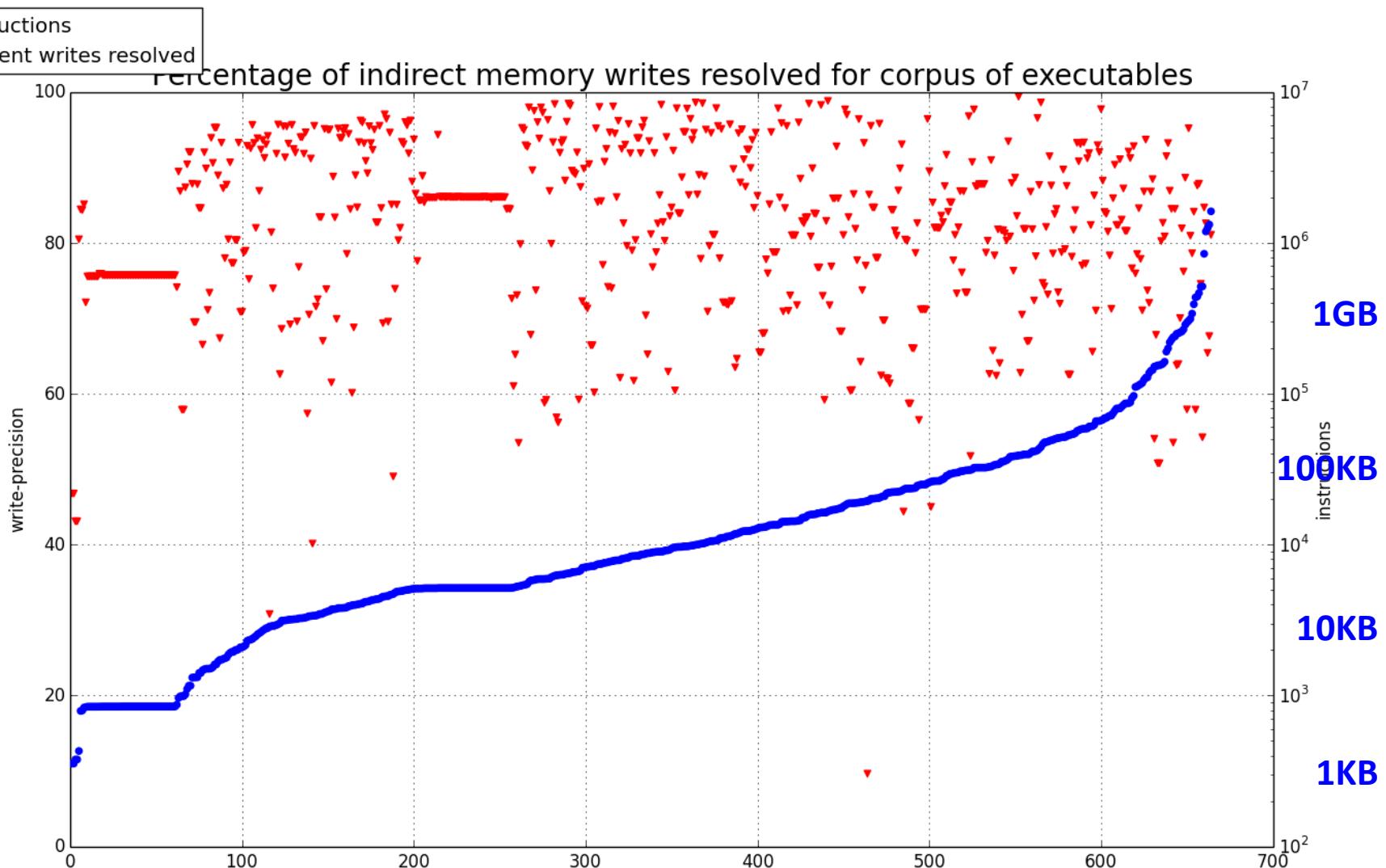
Disassembly:

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

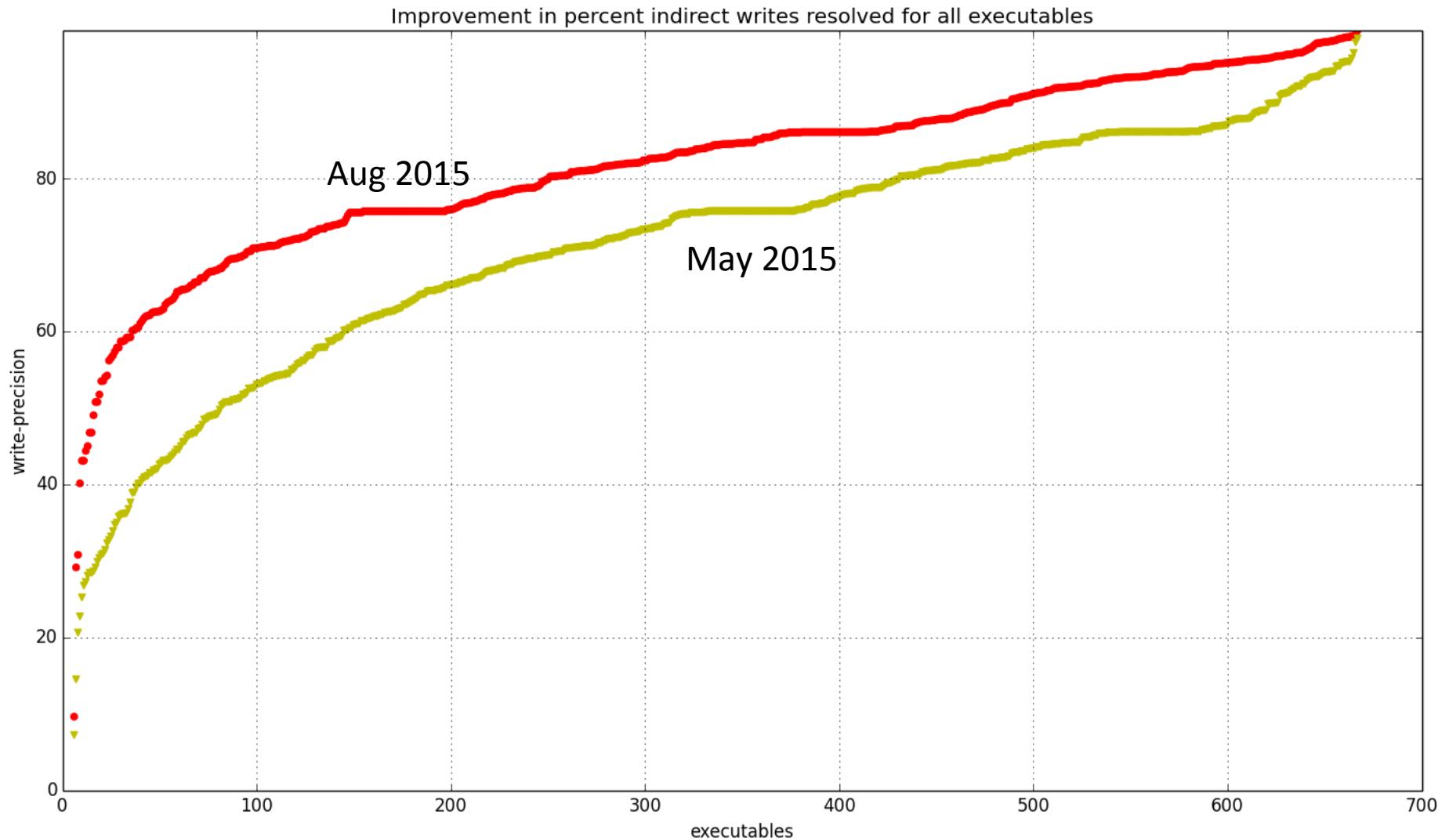
Analysis:

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

Precision: Indirect memory writes resolved



Improvements in Indirect Memory Writes Resolved



CodeHawk Binary Analyzer: Use Cases

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



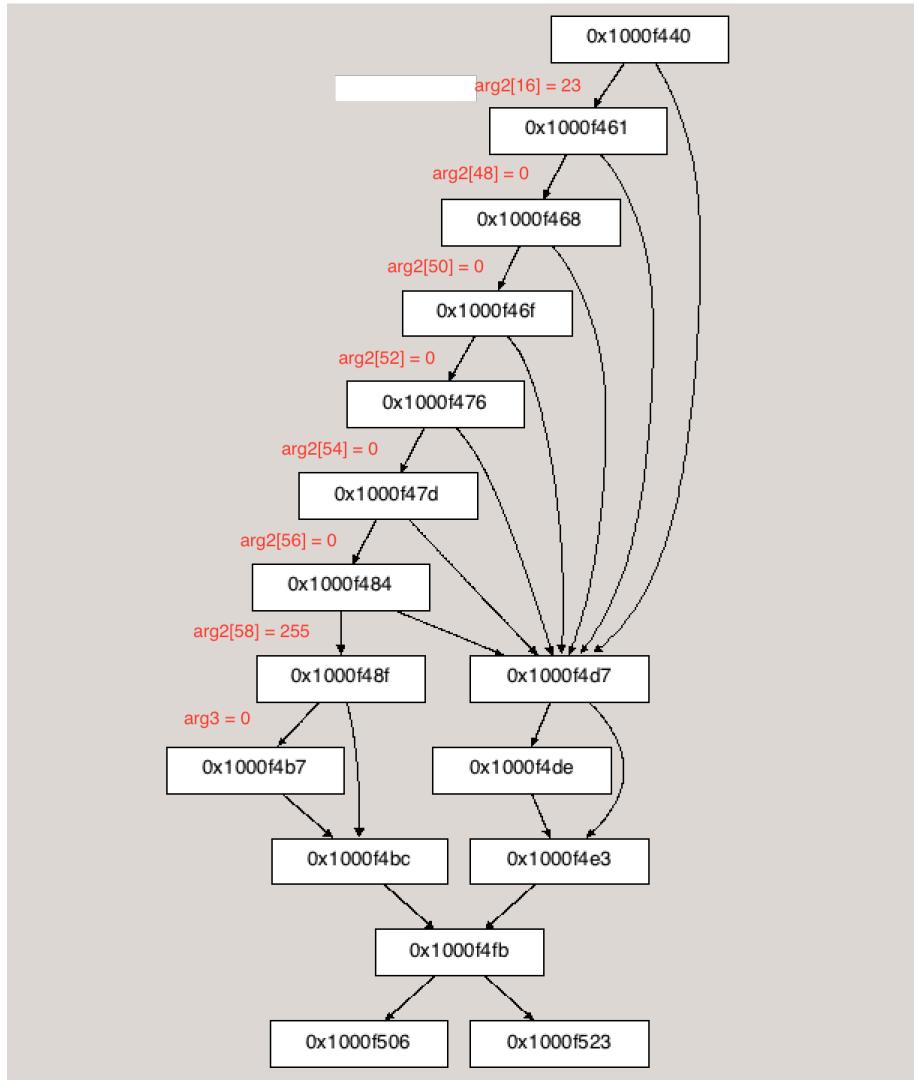
CodeHawk Binary Analyzer: Graphical User Interface



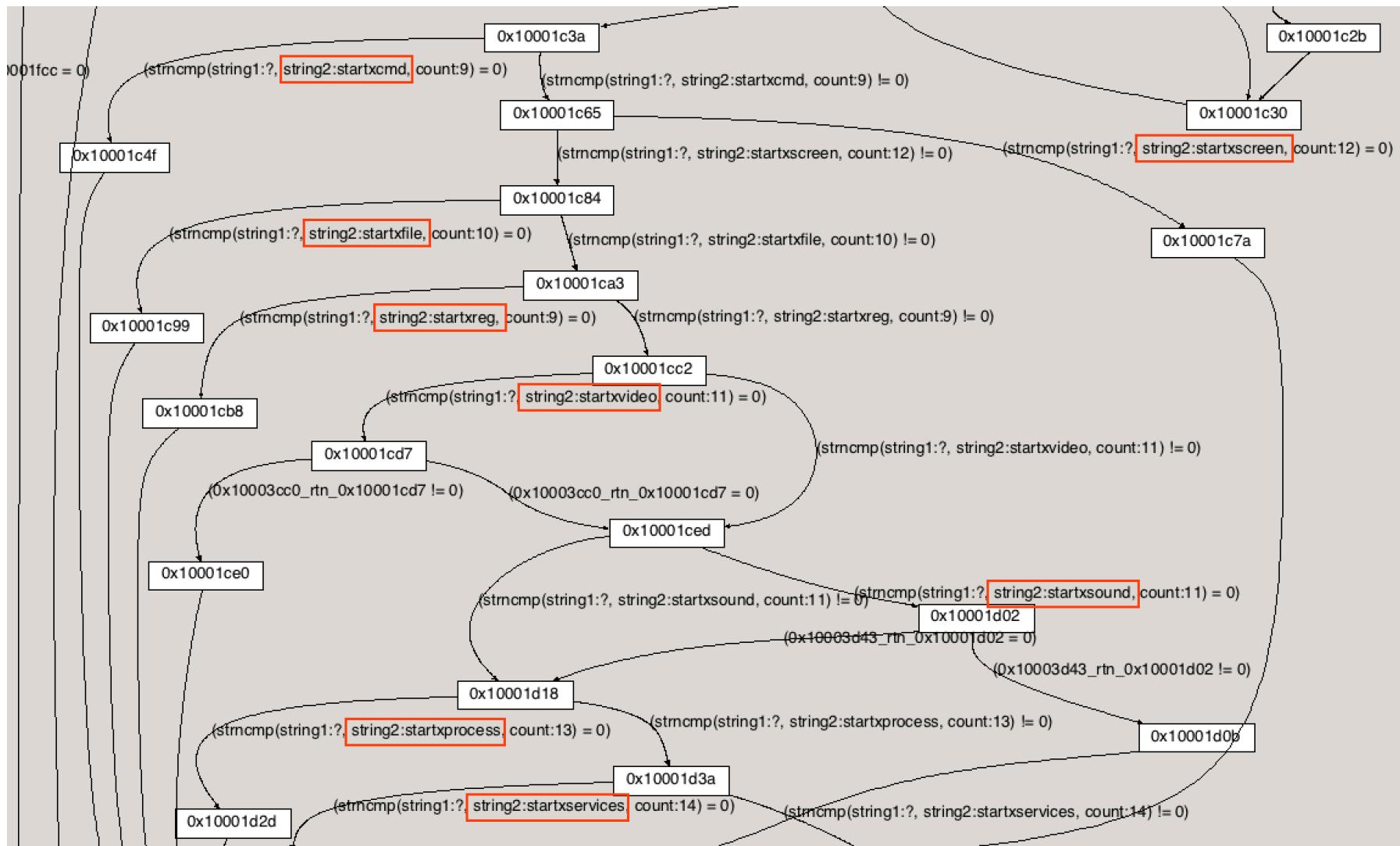
The image displays the CodeHawk Binary Analyzer graphical user interface (GUI) across four windows:

- CodeHawk x86 Binary Analyzer (Main Window):** Shows the assembly code for function 0x406b37. The assembly listing includes columns for address, offset, instruction, and annotations. Annotations show register modifications like "save ebp" and "ebp := esp - 516".
- CFG (Control Flow Graph):** A directed graph showing the control flow between various memory locations (e.g., 0x406b37, 0x406b50, 0x406b76, 0x406bb3, 0x406bb5, 0x406bba). Nodes represent specific memory addresses, and edges represent control flow transitions.
- Register contents for function 0x406b37:** Three tables showing CPU register values at different points in the function. The first table shows initial values (eax=0, ebx=0, etc.). The second table shows values after the first few instructions. The third table shows values near the end of the function, with some registers containing annotations like "(2204 + arg.0004)".
- Register contents for function 0x406b50:** Similar to the previous register table, showing CPU register values for function 0x406b50. It includes annotations such as "no annotation" and "if (Unknown != 0) then goto 0x406b76".

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

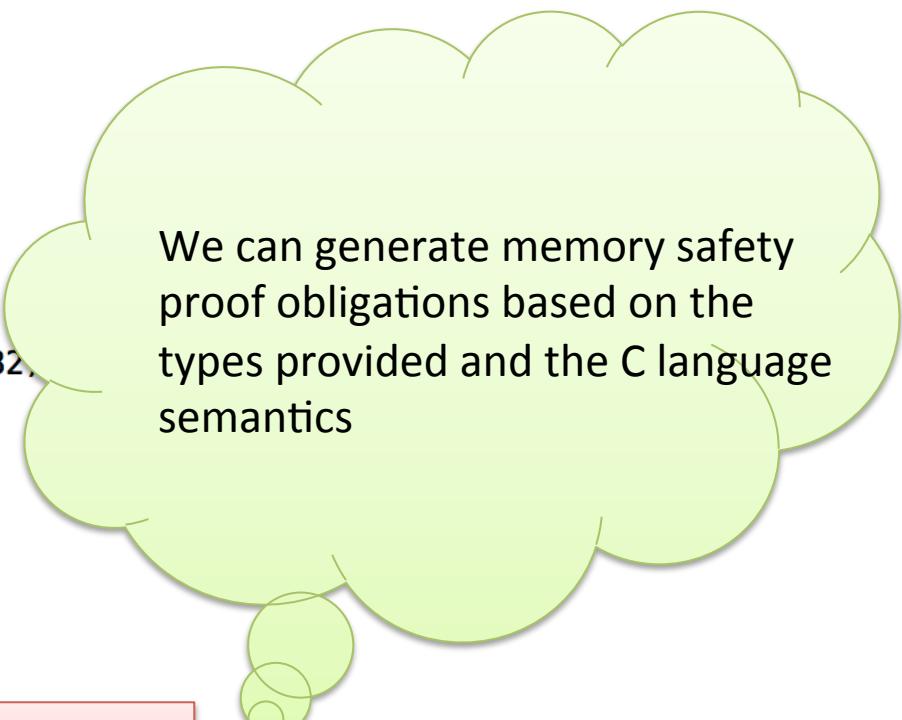
```
unsigned int seedify(char** argv) {
    char key[25];

    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



length(argv[d[1]]) <= 25

Binary Executable Verification

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

We have to infer the size of the strcpy destination buffer

Stackframe for seedify

offset	variable	inferred type	writers/readers
4	arg.0004		

local stack frame

offset	variable	inferred type	writers/readers
-4	var.0004		
-8	var.0008		
-16	var.0016		
-20	var.0020		
-22	var.0022		
-47	var.0047		
-56	var.0056		
-72	var.0072		
-76	var.0076		

-22 var.0022

-47 var.0047

Close

```

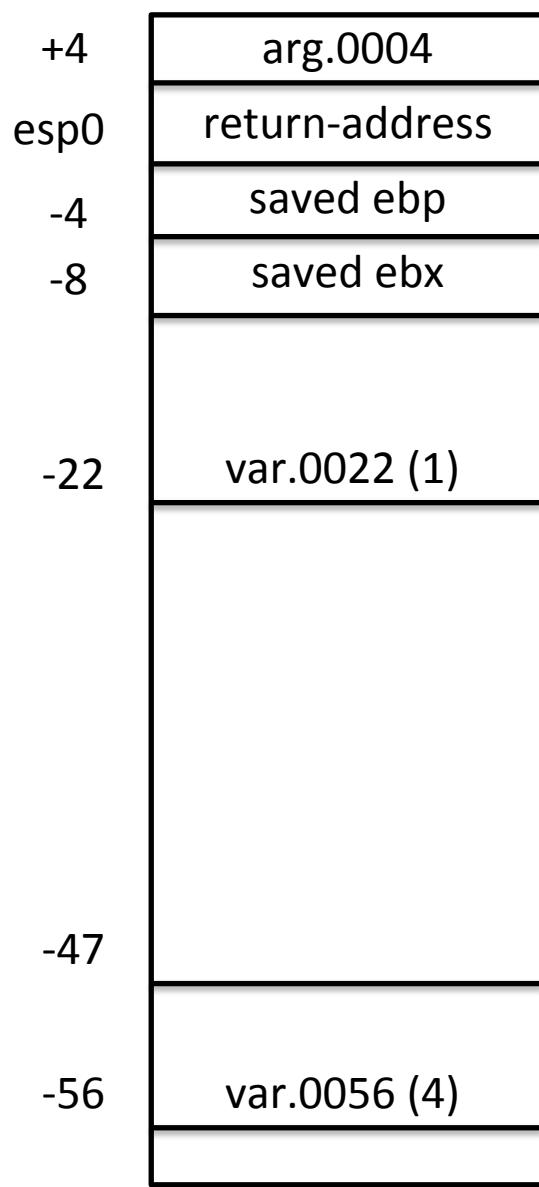
[0] push ebp
[-4] mov esp,ebp
[-4] push ebp
[-8] sub esp,4
[-76] mov -0x4(%ebp),eax
[-76] movsx eax,al
[-76] movsx eax,al
[-76] imul eax,4
[-76] mov -0x4(%ebp),eax
[-76] xchg eax,esi
[-76] mov eax,[esi]
[-76] shl eax,1
[-76] add eax,1
[-76] mov eax,[esi]
[-76] mov 0x4(%esp,,1),eax      [strcpy : src = eax ]
[-76] lea eax,-0x2b(%ebp)       eax := (ebp - 43) = (esp_in - 47)
[-76] mov 0x0(%esp,,1),eax      [strcpy : dest = eax ]

```

X Stack values for function seedify

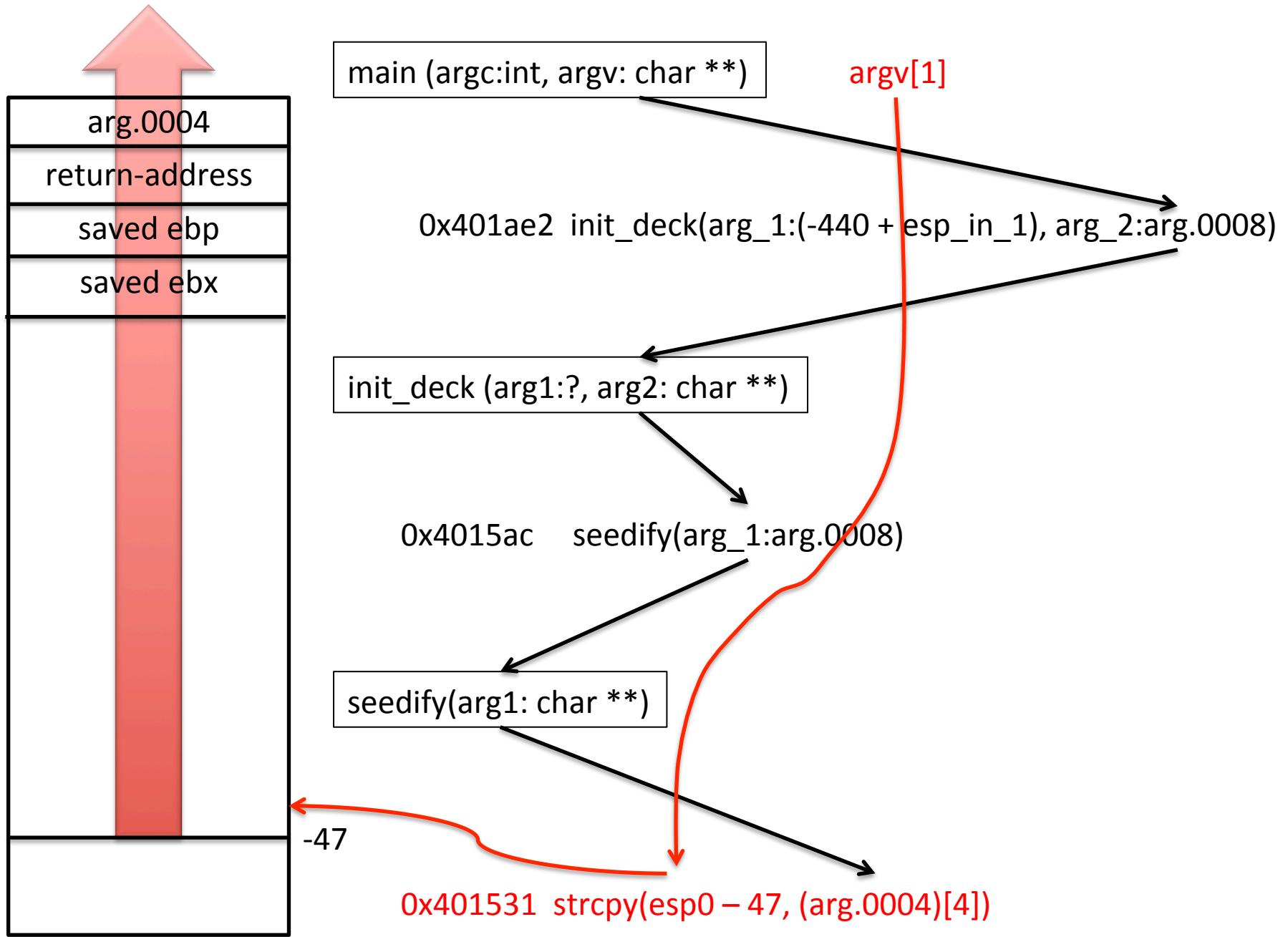
	-76	-72	-56	-47	-22	-20	-16	-8	-4	4	parent
0x4014fb	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	save ebp
0x4014fc	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	ebp := esp = (esp_in - 4)
0x4014fe	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	save ebx
0x4014ff	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	esp := esp - 68 = (esp_in - 76)
0x401502	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	var.0022 := 1
0x401508	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	edx := var.0022 = 1
0x40150c	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	eax := var.0022 = 1
0x401510	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	eax := eax * edx = 1
0x401513	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	var.0056 := 1
0x40151b	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	nop
0x40151c	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	eax := var.0056 = 1
0x40151f	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	eax := eax * 4 = 4
0x401522	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	eax := eax + argv\$1 = (4 + argv\$1)
0x401525	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	eax := argv\$1[4]
0x401527	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[strcpy : src = eax]
0x40152b	[SF:47] (arg.0004)[var.0056:4]	[]	[]	[]	[]	[]	[]	[]	[]	[]	eax := (ebp - 43) = (esp_in - 47)
0x40152e	[SF:47] (arg.0004)[var.0056:4]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[strcpy : dest = eax]
0x401531	SF:-47	(arg.0004)[var.0056:4]	[]	[]	[]	[]	[]	[]	[]	[]	strcpy(dest:&var.0047, src:(arg.0004)[var.0056:4]
0x401536	SF:-47	(arg.0004)[var.0056:4]	[]	se_0x401531_dest	[]	[]	[]	[]	[]	[]	var.0020 := 0
0x40153d	SF:-47	(arg.0004)[var.0056:4]	[]	se_0x401531_dest	[]	[]	[0; 24]	[]	[]	[]	var.0016 := 0
0x401544	SF:-47	(arg.0004)[var.0056:4]	[]	se_0x401531_dest	[]	[]	[0; 24]	[]	[]	[]	goto 0x401570
0x401546	SF:-47	(arg.0004)[var.0056:4]	[]	se_0x401531_dest	[]	[]	[0; 24]	[]	[]	[]	eax := (ebp - 43) = (esp_in - 47)

Vulnerability Research: Where does the input come from?

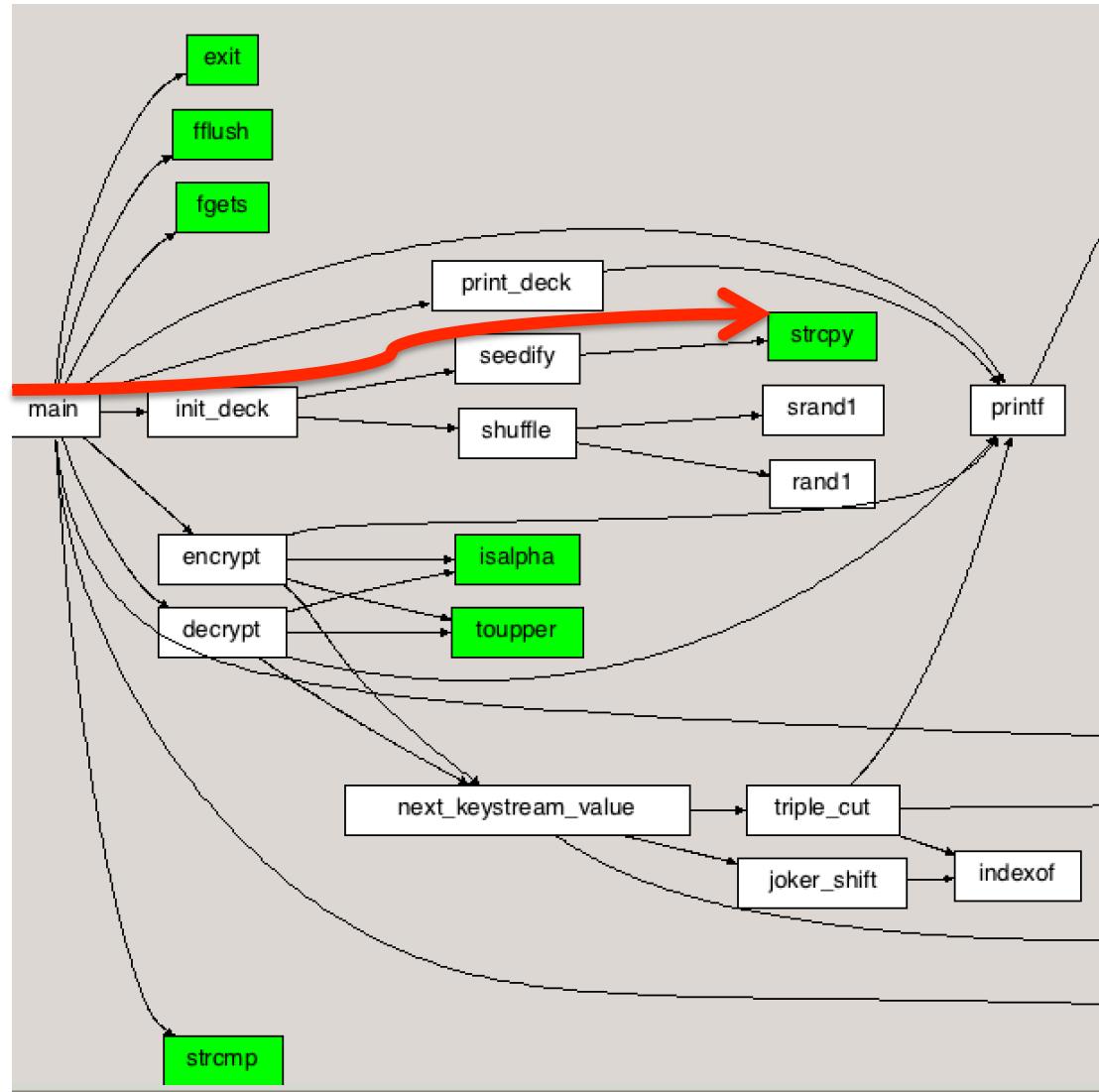


seedify (arg1:char **)

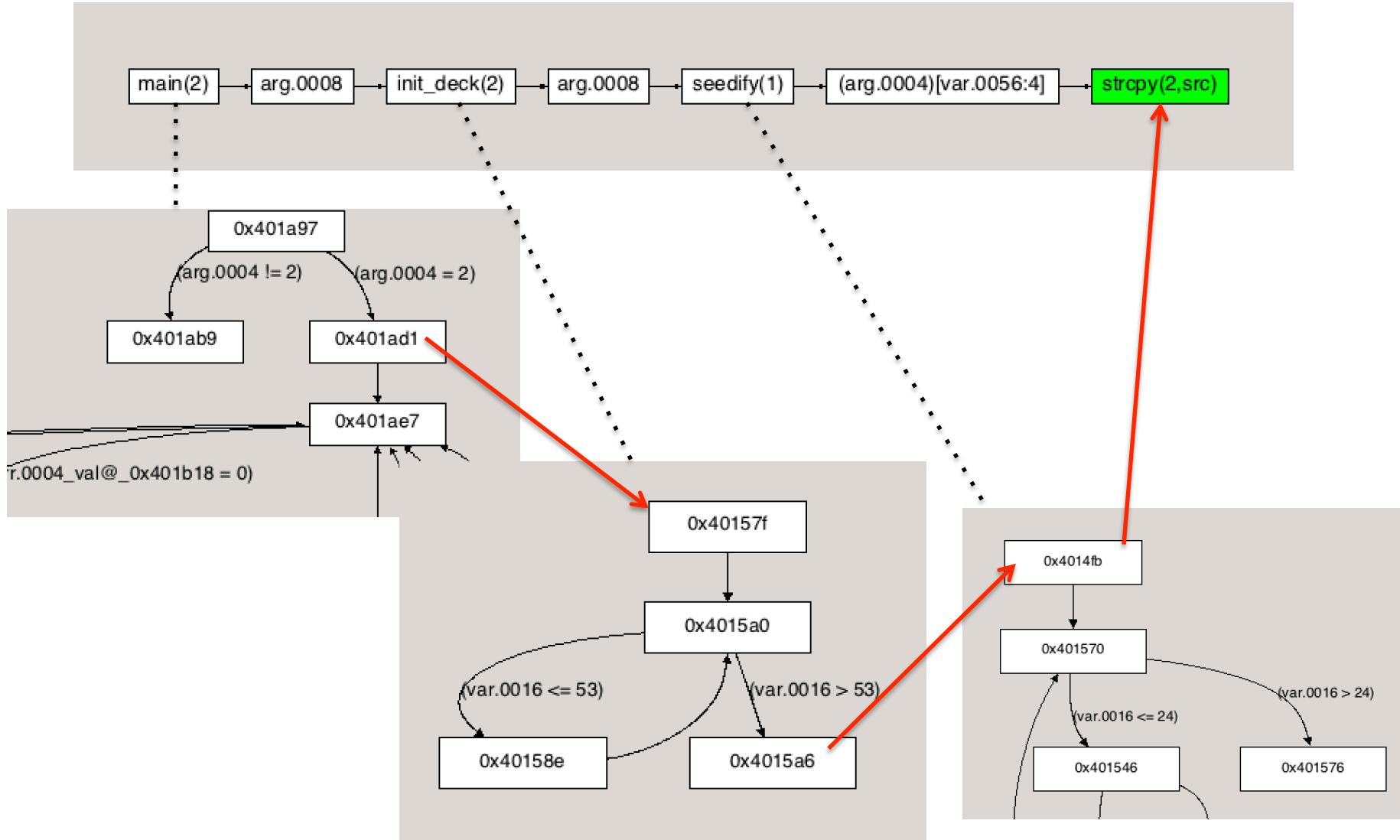
0x401531 strcpy(esp0 - 47, (arg.0004)[4])



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

```

0x401f27 sprintf(buffer:(-84 + esp_in), format:"%d", args:(arg.0012)[6888])
0x401f58
0x402b00
0x402c77
0x402c93
0x402caf
0x4075a3 sprintf(buffer:-712+esp_in, format:"%s:%s", args:(2204 + arg.0004), ?)
0x4075c4
0x407875
0x4088bf
0x4088e5
0x408902
0x40bab0
0x40bad0
0x40c848
0x40c9db
0x40f6b5
0x40f798
0x410102
0x42844f
0x430fec
0x43bed6
0x43bf3c
0x43de29
0x43de50 sprintf(buffer:-1036+esp_in, format:"%s - %s", args:var.0008, ?)
0x43e374
0x43e388
0x43e6cb
0x43efd3
0x43f021
0x43f021
0x43f21c
0x43f272

sprintf(buffer:(-48 + esp_in), format:"%d", args:var.0008)
sprintf(buffer:(-84 + esp_in), format:"%d", args:var.0100)
sprintf(buffer:(-84 + esp_in), format:"%g", args:arg.0012)
sprintf(buffer:(-16 + esp_in), format:"<%02X>", args:arg.0004)
sprintf(buffer:(esp_in - 104), format:4559408, args:var.0128)
sprintf(buffer:(-24 + esp_in), format:"XX", args:?)
sprintf(buffer:(-24 + esp_in), format:"%02x", args:var.0120)
sprintf(buffer:(-712 + esp_in), format:"%s:%s", args:(2204 + arg.0004))
sprintf(buffer:(-456 + esp_in), format:"Proxy-Authorization: Basic ", args:?)
sprintf(buffer:(-36 + esp_in), format:"%i", args:arg.0008)
sprintf(buffer:(-88 + esp_in), format:"%d", args:var.0616)
sprintf(buffer:(-40 + esp_in), format:"Colour%d", args:edi)
sprintf(buffer:(-40 + esp_in), format:"Wordness%d", args:?)
sprintf(buffer:(-100 + esp_in), format:"%dx%d", args:arg.0008)
sprintf(buffer:(-108 + esp_in), format:"%d ". aras:bianum_bitcount rtn 0x428440)

sprintf(buffer:(-88 + esp_in), format:"\u0d", args:?)
sprintf(buffer:(-444 + esp_in), format:"Unable to play sound file\n%s\nUsing default sound instead", args:4680576)
sprintf(buffer:(-104 + esp_in), format:".70s Sound Error", args:"PuTTY")
sprintf(buffer:(-104 + esp_in), format:".70s (inactive)", args:"PuTTY")
sprintf(buffer:(-104 + esp_in), format:".70s Fatal Error", args:"PuTTY")
sprintf(buffer:(-216 + esp_in), format:".70s Fatal Error", args:"PuTTY")
sprintf(buffer:(-104 + esp_in), format:".70s Fatal Error", args:"PuTTY")
sprintf(buffer:(-1036 + esp_in), format:"Unable to open connection to\n%.800s\n%s", args:cfg_dest_rtn_0x43f209)
sprintf(buffer:(-1036 + esp_in), format:"%s - %s", args:var.0008)

```



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