Identifying Variables in x86 Executables

Gogul Balakrishnan
Thomas Reps
University of Wisconsin
Motivation

• Code-inspection tools for security analysts
  - dependence-based navigation ("code surfing")

• Analyses for identifying
  - security vulnerabilities and bugs
  - malicious code
  - commonalities and differences

• Platform for
  - code obfuscation and de-obfuscation
  - de-compilation
  - installation of protection mechanisms
  - remediation of security vulnerabilities
Why Executables?

• Reflects actual behaviors that may arise
• Allows platform-specific artifacts to be taken into account
  - memory layout
  - register usage
  - execution order
  - compiler bugs
  - Thompson-style attack
• Source code hides the low-level (actual) behaviors that implement high-level abstractions
• Source-code analyses typically make unsafe assumptions (e.g., that the program is ANSI-C compliant)
  - loss of fidelity can allow vulnerabilities to escape notice
int callee(int a, int b) {
    int local;
    if (local == 5) return 1;
    else return 2;
}

int main() {
    int c = 5;
    int d = 7;
    int v = callee(c, d);
    // What is the value of v here?
    return 0;
}

Answer: 1
(for the Microsoft compiler)
Initial estimate of
• code vs. data
• procedures
• call sites
• malloc sites
CodeSurfer/x86 Architecture

IR Recovery
- fleshed-out CFGs
- fleshed-out call graph
- used, killed, may-killed variables for CFG nodes
- points-to sets
- reports of violations
- [variables]
- [types: base types, pointer types, structs, and classes]
Scope

• Programs that conform to a “standard compilation model”
  - procedures
  - activation records
  - global data region
  - heap, etc.

• Report violations
  - violations of stack protocol
  - return address modified within procedure
Technical Challenges

- Distinguishing between code and data
- Identifying variables

- Identifying parameters
- Resolving indirect jumps
- Resolving indirect calls
- Identifying may-aliases
Technical Challenges

- Distinguishing between code and data
- Identifying variables
- Identifying parameters
- Resolving indirect jumps
- Resolving indirect calls
- Identifying may-aliases
int arrVal = 0, *pArray2;

int main() {
    int i, a[10], *p;
    /* Initialize pointers */
    pArray2 = &a[2];
    p = &a[0];
    /* Initialize Array */
    for (i = 0; i < 10; ++i) {
        *p = arrVal;
        p++;
    }
    /* Return a[2] */
    return *pArray2;
}
int arrVal=0, *pArray2;

int main() {
    int i, a[10], *p;
    /* Initialize pointers */
    pArray2 = &a[2];
    p = &a[0];
    /* Initialize Array */
    for(i = 0; i<10; ++i) {
        *p = arrVal;
        p++;
    }
    /* Return a[2] */
    return *pArray2;
}
Running Example - Address Space

; ebx ⇔ variable i
; ecx ⇔ variable p

sub esp, 40       ;adjust stack
lea edx, [esp+8]  ;mov [8], edx ;pArray2=&a[2]

loc_9:
mov [ecx], edx ;*p=arrVal
add ecx, 4      ;p++
inc ebx; i++
cmp ebx, 10     ;i<10?
jl short loc_9;

mov edi, [8]    ;
mov eax, [edi] ;return *pArray2
add esp, 40
retn
Running Example - Address Space

Data local to main (Activation Record)

No debugging information

Global data

`; ebx ⇔ variable i
; ecx ⇔ variable p`

```
sub esp, 40 ; adjust stack
lea edx, [esp+8] ;
mov [8], edx ; pArray2=&a[2]
lea ecx, [esp] ; p=&a[0]
mov edx, [4] ;

loc_9:
  mov [ecx], edx ; *p=arrVal
  add ecx, 4 ; p++
  inc ebx ; i++
  cmp ebx, 10 ; i<10?
  jl short loc_9 ;

mov edi, [8] ;
```

```
mov eax, [edi] ; return *pArray2
add esp, 40
retn
```
Identifying Variables

• An abstraction of concrete memory configurations
  - Memory regions

• Infer layout of memory regions
  - A-locs (like variables)
Memory Regions

- An abstraction of concrete memory configurations
  - Idea: group similar runtime addresses
  - e.g., collapse the runtime ARs for each procedure, malloc-sites, global data
Example - Memory Regions

```
; ebx ⇔ variable i
; ecx ⇔ variable p

sub esp, 40       ;adjust stack
lea edx, [esp+8]  ;mov [8], edx ;pArray2=&a[2]
lea ecx, [esp]    ;p=&a[0]
mov edx, [4] ;

loc_9:
  mov [ecx], edx ;*p=arrVal
  add ecx, 4 ;p++
  inc ebx ;i++
  cmp ebx, 10 ;i<10?
  jl short loc_9 ;

mov edi, [8] ;
mov eax, [edi] ;return *pArray2
add esp, 40
retn
```

Region for main

Global Region

(GL, 4)

(GL, 12)

ret_main

(main, 0)

(main, -40)

Example – Memory Regions
Infer Layout of Memory Regions

- Data-layout known at assembly/compile time
  - some variables held in registers
  - global variables $\rightarrow$ absolute addresses
  - local variables $\rightarrow$ offsets in stack frame

- A-locs
  - locations between consecutive addresses
  - locations between consecutive offsets
  - registers
Example - A-locs

; ebx ⇔ variable i
; ecx ⇔ variable p

sub esp, 40       ;adjust stack
lea edx, [esp+8]  ;pArray2=&a[2]
lea ecx, [esp]    ;p=&a[0]
mov edx, [4]      ;
loc_9:
mov [ecx], edx ;*p=arrVal
add ecx, 4       ;p++
inc ebx          ;i++
cmp ebx, 10      ;i<10?
jl short loc_9   ;
mov edi, [8]     ;
mov eax, [edi]   ;return *pArray2
add esp, 40
retn

Example – A-locs
Example – A-locs

; ebx ⇔ variable i
; ecx ⇔ variable p

sub esp, 40 ;adjust stack
lea edx, [esp+8] ;pArray2=&a[2]
mov [8], edx ;p=&a[0]
lea ecx, [esp] ;p=&a[0]
mov edx, [4] ;

loc_9:
    mov [ecx], edx ;*p=arrVal
    add ecx, 4 ;p++
    inc ebx ;i++
    cmp ebx, 10 ;i<10?
    jl short loc_9 ;

mov edi, [8] ;
mov eax, [edi] ;return *pArray2
add esp, 40
retn
Example – A-locs

Region for main

(main, 0) → ret_main

(main, -32) → mainv_40

(main, -40) → mainv_32

Global Region

(GL, 12) → mem_8

(GL, 8) → mem_4

(GL, 4) → mainv_40

; ebx \leftrightarrow variable i

; ecx \leftrightarrow variable p

sub esp, 40 ;adjust stack
lea edx, &mainv_32;
mov mem_8, edx ;pArray2=&a[2]
lea ecx, &mainv_40;p=&a[0]
mov edx, mem_4

loc_9:
    mov [ecx], edx ;*p=arrVal
    add ecx, 4 ;p++
    inc ebx ;i++
    cmp ebx, 10 ;i<10?
    jl short loc_9 ;

    mov edi, mem_8 ;
    mov eax, [edi] ;return *pArray2

add esp, 40
ret
Better Identification of Variables

- **IDAPro A-locs**
  - Based on explicitly specified addresses/offsets
- **VSA provides access patterns for indirect operands**
  - \texttt{ecx} → (⊥, 4[0,9]-40)

```
loc_9:
    mov [ecx], edx ;*p=arrVal
    add ecx, 4 ;p++
    inc ebx ;i++
    cmp ebx, 10 ;i<10?
    jl short loc_9 ;

    mov edi, [4]
    mov eax, [edi] ;return *pArray2
    add esp, 40
    retn
```
Aggregate Structure Identification

• Partition aggregates according to the program’s memory-access patterns
  - original motivation: Y2K [Ramalingam et al. POPL 99]

• Uses in our context
  - improved identification of variables
    • identifies a better set of a-locs
      ⇒ better IR ⇒ fewer false alarms
  - recovery of type information
    • identifies structs and arrays
    • propagates type information from known parameter types (system calls & library functions)
      ⇒ better de-compilation
Aggregate Structure Identification

; ebx ⇔ variable i
; ecx ⇔ variable p

sub esp, 40 ; adjust stack
lea edx, [esp+8] ;
mov [4], edx ;pArray2=&a[2]
lea ecx, [esp] ;p=&a[0]
mov edx, [0] ;

loc_9:
    mov [ecx], edx ;*p=arrVal
    add ecx, 4 ;p++
    inc ebx ;i++
cmp ebx, 10 ;i<10?
    jl short loc_9 ;

mov edi, [4] ;
mov eax, [edi] ; return *pArray2
add esp, 40
retn
Aggregate Structure Identification

; ebx ⇔ variable i
; ecx ⇔ variable p

sub esp, 40       ;adjust stack
lea edx, [esp+8]  ;
mov [4], edx      ;pArray2=&a[2]
lea ecx, [esp]    ;p=&a[0]
mov edx, [0]      ;

loc_9:
mov [ecx], edx    ;*p=arrVal
add ecx, 4        ;p++
inc ebx           ;i++
cmp ebx, 10       ;i<10?
jl short loc_9    ;
mov edi, [4]      ;
mov eax, [edi]    ;return *pArray2
add esp, 40
retn
Aggregate Structure Identification

ASI: two arrays; one scalar

AR[-40:-1]

40

2⊗

32

1⊗

7⊗

4

4

4

; ebx ⇔ variable i
; ecx ⇔ variable p

sub esp, 40 ;adjust stack
lea edx, [esp+8] ;
mov [4], edx ;pArray2=&a[2]
lea ecx, [esp] ;p=&a[0]
mov edx, [0] ;

loc_9:

mov [ecx], edx ;*p=arrVal
add ecx, 4 ;p++
inc ebx ;i++
cmp ebx, 10 ;i<10?
jl short loc_9 ;

mov edi, [4] ;
mov eax, [edi] ;return *pArray2
add esp, 40
retn
Aggregate Structure Identification

AS1: two arrays; one scalar

Region for main

IDA Pro
one 8-byte a-loc
one 32-byte a-loc
Aggregate Structure Identification

High level type:

```c
struct {
    int a[2];
    int b;
    int c[7];
};
```

ASI: two arrays; one scalar
CodeSurfer/x86 Architecture

Technical Challenges
- Distinguishing between code and data
- Identifying variables
- Identifying parameters
- Resolving indirect jumps
- Resolving indirect calls
- Identifying may-aliases

- fleshed-out CFGs
- fleshed-out call graph
- used, killed, may-killed variables for CFG nodes
- points-to sets
- reports of violations
CodeSurfer/x86 Architecture

Binary

IDA Pro
- Parse Binary
- Build CFGs

Connector
- VSA
- ASI

CodeSurfer
- Build SDG
- Browse

Technical Challenges
- Distinguishing between code and data
- Identifying variables

- Identifying parameters
- Resolving indirect jumps
- Resolving indirect calls
- Identifying may-aliases

- fleshed-out CFGs
- fleshed-out call graph
- used, killed, may-killed variables for CFG nodes
- points-to sets
- reports of violations

Security Analyzers
Decompiler
Binary Rewriter
User Scripts
CodeSurfer/x86 Architecture

Binary
- Parse Binary
- Build CFGs

Connector
- VSA
- ASI

CodeSurfer
- Build SDG
- Browse

Security Analyzers
- Decompiler
- Binary Rewriter
- User Scripts

Technical Challenges
- Distinguishing between code and data
- Identifying variables
- Identifying parameters
- Resolving indirect jumps
- Resolving indirect calls
- Identifying may-aliases

- fleshed-out CFGs
- fleshed-out call graph
- used, killed, may-killed variables for CFG nodes
- points-to sets
- reports of violations
Wrap Up

- **Code-inspection tools for security analysts**
  - dependence-based navigation ("code surfing")

- **Analyses for identifying**
  - security vulnerabilities and bugs
  - malicious code
  - commonalities and differences

- **Platform for**
  - code obfuscation and de-obfuscation
  - de-compilation
  - installation of protection mechanisms
  - remediation of security vulnerabilities
Identifying Variables in x86 Executables

Gogul Balakrishnan
Thomas Reps
University of Wisconsin