# On the practical study of IR-based Value Set Analysis

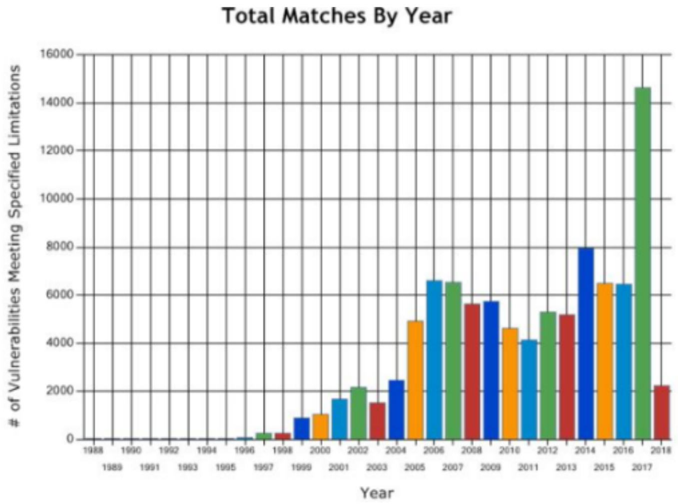
**Hyunki Kim**
Graduate School of Information Security
KAIST

December 16, 2019

# Motivation

- U.S. gov, software vulnerabilities are on the rise (2018)
  - Almost software vulnerabilities are occurred by binary variables

- 2019 CWE top 25 related to variables
  - Improper input validation
  - Use after free
  - Integer overflow
  - …



Statistics on software vulnerability reported since 1988 – 2018/3(NIST)

# Previous research

- Variable recovery technique
  - Executable Analysis using Abstract Interpretation with Circular Linear Progressions (2007)
    - CLPs (Circular Linear Progressions)
    - Rathijit Sen and Y. N. Srikant
  - WYSINWYX: WHAT YOU SEE IS NOT WHAT YOU EXECUTE (2007)
    - VSA (Value Set Analysis)
    - Gogul Balakrishnan
    - Popular method for variables recovery (by Sec'19)
    - Strided-interval expression with memory region and abstract location
  - Analyzing Stripped Device-Driver Executables (2008)
    - Gogul Balakrishnan and Thomas Reps
  - …

# Previous research

- Variable recovery technique
  - Executable Analysis using Abstract Interpretation with Circular Linear Progressions (2007)
    - CLPs (Circular Linear Progressions)
    - Rathijit Sen and Y. N. Srikant
  - **WYSINWYX: WHAT YOU SEE IS NOT WHAT YOU EXECUTE (2007)**
    - **VSA (Value Set Analysis)**
    - **Gogul Balakrishnan**
    - **Popular method for variables recovery (by Sec'19)**
    - **Strided-interval expression with memory region and abstract location**
  - Analyzing Stripped Device-Driver Executables (2008)
    - Gogul Balakrishnan and Thomas Reps
  - …

# Goals

- ❖ Variables recovery
  - – Methodology : Value Set Analysis
    - ▪ Close source
  - – Framework : B2R2 (Version : May, 2019)
    - ▪ State of the art binary analysis tool
    - ▪ 2019 NDSS
      Binary Analysis Research workshop
      Best Paper Award

| | | | Lifter Design | | | IR Characteristics | | | | | | Architecture Support | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Tool | IR Name | Programming Language | Pure Parallelism | IR Optimization | AST Construction | Metadata Embedding | Explicit[5] | Self-Contained[5] | Hash-consed IR Support | x86 SIMD Support | Big Integer Splitting | x86 | x86-64 | ARMv7 | ARMv8 | Thumb | MIPS32 | MIPS64 | PPC32 | PPC64 |
| angr [43] | VEX[2] (Valgrind [37]) | C & Python | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| BAP [13] | BIL | OCaml[4] | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |
| BINSEC [19] | DBA | OCaml | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| BinNavi [20] | REIL | Java | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| BitBlaze [44] | Vine | C & OCaml | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Insight [22] | Microcode[3] | C++ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Jakstab [31] | SSL | Java | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Miasm [15] | Miasm IR | C & Python | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ |
| radare2 [4] | ESIL [1] | C | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ |
| rev.ng [18] | LLVM | C++ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| B2R2★ | LowUIR | F# | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |

Existing open-source tools for binary analysis
(B2R2: Building an Efficient Front-End for Binary Analysis)



SysSec
System Security Lab

# Binary analysis

Static analysis

- Code-based flow analysis
- Limited analysis time

→ Abstract interpretation + interval analysis

- Using approximation with intervals

→ Strided-interval analysis

- Using stride in interval analysis

→ Value set analysis

# Binary analysis

**Static analysis**

- **Code-based flow analysis**
- **Limited analysis time**

→ Abstract interpretation + interval analysis

- Using approximation with intervals

→ Strided-interval analysis

- Using stride in interval analysis

→ Value set analysis

```
sub esp, 44
lea eax, [esp+8]
mov [esp+0], eax
mov edx, 0
inc edx
add  ebx,  2
…
```

# Binary analysis

Static analysis

- Code-based flow analysis
- Limited analysis time

→ **Abstract interpretation + interval analysis**

- **Using approximation with intervals**

→ Strided-interval analysis

- Using stride in interval analysis

→ Value set analysis

**sub esp, 44**
**lea eax, [esp+8]**
**mov [esp+0], eax**
**mov edx, 0**
**inc edx**
**add ebx, 2**
**…**

---

**edx = 1**
**0x80…02 <= ebx <= +inf**
**…**

# Binary analysis

Static analysis

- Code-based flow analysis
- Limited analysis time

→ **Abstract interpretation + interval analysis**

- **Using approximation with intervals**

→ Strided-interval analysis

- Using stride in interval analysis

→ Value set analysis

```
sub esp, 44
lea eax, [esp+8]
mov [esp+0], eax
mov edx, 0
inc edx
add  ebx,  2
...
```

**edx = 1**

**0x80…02 <= ebx <= +inf**

**…**

{ a = 1 or 7 | 1<=a<=7 }

# Binary analysis

Static analysis

- Code-based flow analysis
- Limited analysis time

→ Abstract interpretation + interval analysis

- Using approximation with intervals

→ **Strided-interval analysis**

- **Using stride in interval analysis**

→ Value set analysis

```
sub esp, 44
lea eax, [esp+8]
mov [esp+0], eax
mov edx, 0
inc edx
add  ebx,  2
...
```

```
edx = 0[1,1]
ebx = 1[0x80…02, +inf]
...
```

# Binary analysis

Static analysis

- Code-based flow analysis
- Limited analysis time

→ Abstract interpretation + interval analysis

- Using approximation with intervals

→ **Strided-interval analysis**

- **Using stride in interval analysis**

→ Value set analysis

**sub esp, 44**
**lea eax, [esp+8]**
**mov [esp+0], eax**
**mov edx, 0**
**inc edx**
**add ebx, 2**
**…**

**edx = 0[1,1]**
**ebx = 1[0x80…02, +inf]**
**…**

{ a = 1 or 7 | a = 6[1,7] }

# Binary analysis

Static analysis

- Code-based flow analysis
- Limited analysis time

→ Abstract interpretation + interval analysis

- Using approximation with intervals

→ Strided-interval analysis

- Using stride in interval analysis

→ **Value set analysis**

```
sub esp, 44
lea eax, [esp+8]
mov [esp+0], eax
mov edx, 0
inc edx
add  ebx,  2
...
```

**Register(“esp”)**
**[Global SI:1[0xFFFFFFBC, 0xFFFFFFBC]]**

# Binary analysis

Static analysis

- Code-based flow analysis
- Limited analysis time

→ Abstract interpretation + interval analysis

- Using approximation with intervals

→ Strided-interval analysis

- Using stride in interval analysis

→ **Value set analysis**

```
sub esp, 44
lea eax, [esp+8]
mov [esp+0], eax
mov edx, 0
inc edx
add  ebx,  2
...
```

**Register("esp") [Global SI:1[0xFFFFFFBC, 0xFFFFFFBC]]**
**Register("eax") [Global SI:1[-inf, +inf]]**

# Binary analysis

Static analysis

- Code-based flow analysis
- Limited analysis time

→ Abstract interpretation + interval analysis

- Using approximation with intervals

→ Strided-interval analysis

- Using stride in interval analysis

→ **Value set analysis**

```
sub esp, 44
lea eax, [esp+8]
mov [esp+0], eax
mov edx, 0
inc edx
add  ebx,  2
...
```

**Register("esp")**
**[Global SI:1[0xFFFFFFBC,**
**0xFFFFFFBC]]**
**Register("eax") [Global SI:1[-inf, +inf]]**
**Register("edx") [Global SI:0[0,0]]**

# Binary analysis

Static analysis

- Code-based flow analysis
- Limited analysis time

→ Abstract interpretation + interval analysis

- Using approximation with intervals

→ Strided-interval analysis

- Using stride in interval analysis

→ **Value set analysis**

```
sub esp, 44
lea eax, [esp+8]
mov [esp+0], eax
mov edx, 0
inc edx
add  ebx,  2
…
```

Register("esp")
[Global SI:1[0xFFFFFFBC,
0xFFFFFFBC]]
Register("eax") [Global SI:1[-inf, +inf]]
Register("edx") [Global SI:0[1,1]]

# Value Set Analysis (VSA)

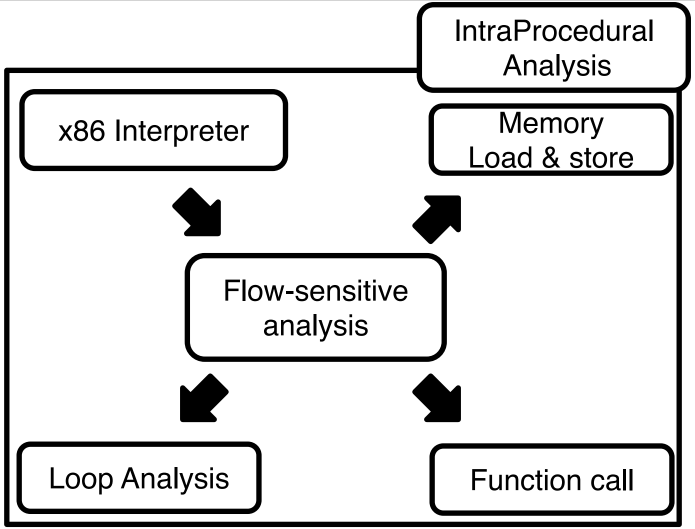
| A-Loc | Memory region | Strided Interval |
|---|---|---|
| Memory | Stack | Min |
| Register | Global | Max |
| Flag | Heap | Stride |

Value-set (VS)

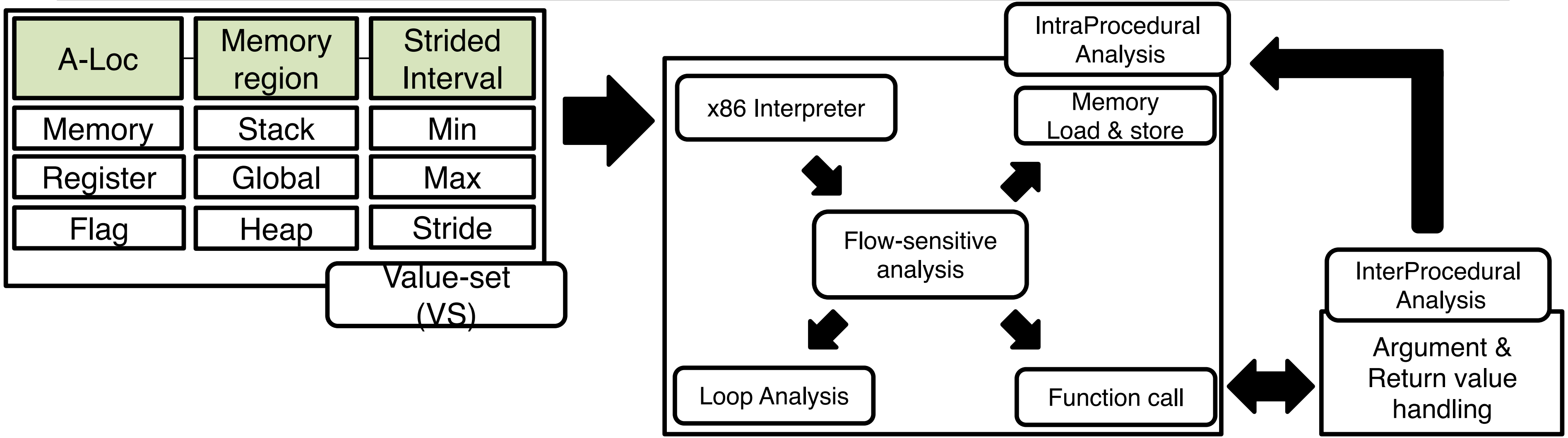
# Value Set Analysis (VSA)

| A-Loc | Memory region | Strided Interval |
|---|---|---|
| Memory | Stack | Min |
| Register | Global | Max |
| Flag | Heap | Stride |

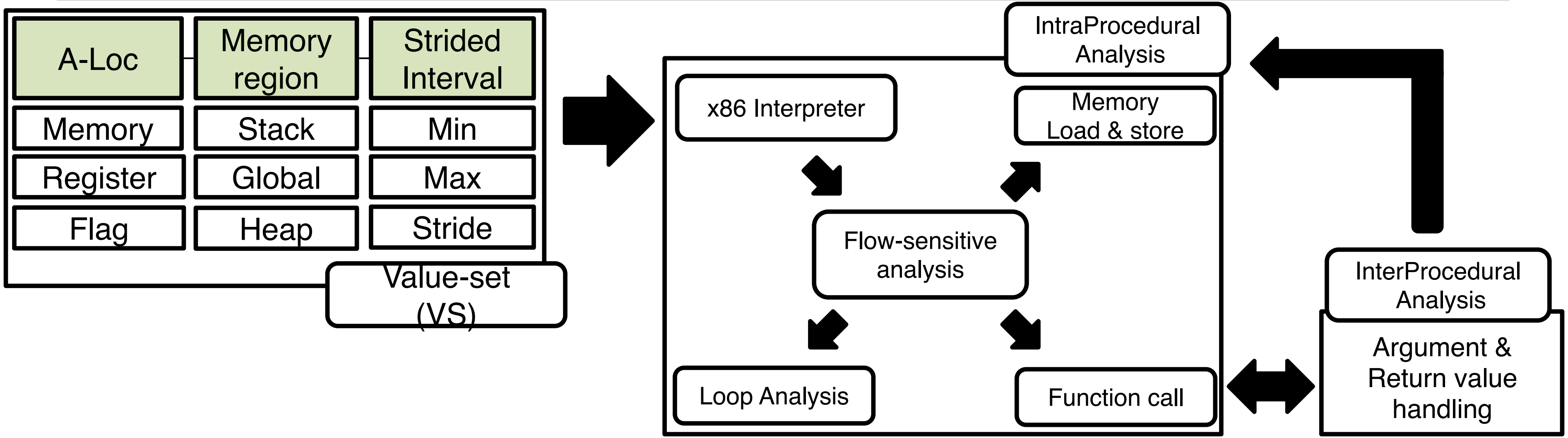
Value-set (VS)



# Value Set Analysis (VSA)



# Value Set Analysis (VSA)



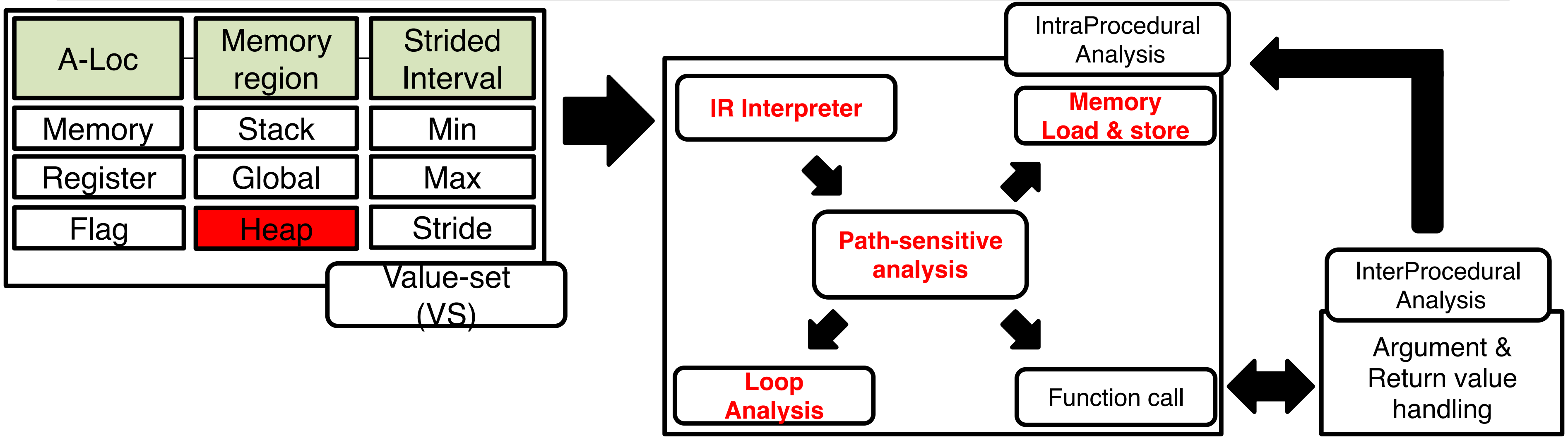

```
mov ebx,  eax
add  ebx,  4
        ...
```

Assembly Instruction

**MemVar("main",-20L) [Stack SI:1[0x1, 0x4]]**
**Register("RSP") [Global SI:1[0x20, 0x20]]**
**Flag("OF") [Global SI: 0[0x1, 0x1]]**
**...**

Value Set Analysis

# Value Set Analysis (VSA)




**Def (PCVar (62, 28), Num 0x33:I64)**
**Def (RegVar (64, 1, "RSP", 1),**
**BinOp (ADD, 64, Num 0x1:I64, Num 0x2:I64))**
**...**

**B2R2 IR Instruction**

**MemVar("main",-20L) [Stack SI:1[0x1, 0x4]]**
**Register("RSP") [Global SI:1[0x20, 0x20]]**
**Flag("OF") [Global SI: 0[0x1, 0x1]]**
**...**

Value Set Analysis

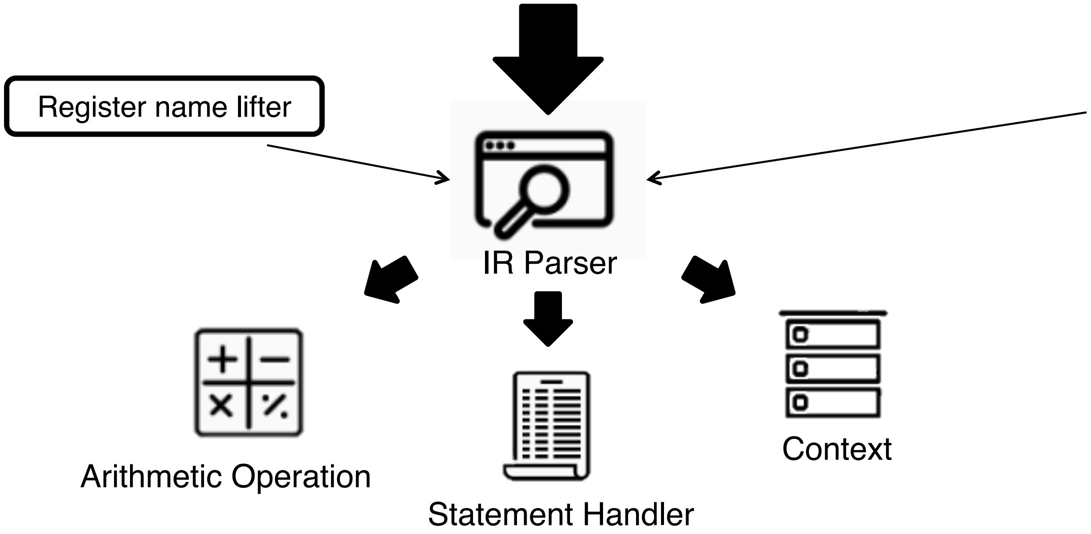
# IR Interpreter

- Pros and Cons
  - Pros
    - Various language supporting
      - Register name lifting
    - Hardware specific details are encoded in assembly
      - Flag instruction
    - Simple operation
  - Cons
    - Too much instruction
      - One assembly code → several IR code

# IR Interpreter

```
Def (PCVar (62, 28), Num 0x33:I64)
Def (RegVar (64, 1, "RSP", 1),
    BinOp (ADD, 64, Num 0x1:I64, Num 0x2:I64))
          ...
```

B2R2 IR Instruction




| | | | |
|---|---|---|---|
| METADATA | $\mu$ | ::= | ExprInfo * ConsInfo |
| | | \| | ExprInfo |
| ENDIAN | $\epsilon$ | ::= | BEndian \| LEndian |
| UNOP | $\Diamond_u$ | ::= | NEG \| NOT |
| BINOP | $\Diamond_b$ | ::= | ADD \| SUB \| MUL \| DIV \| SDIV \| MOD \| SMOD \| SHL \| SHR \| SAR \| AND \| OR \| XOR \| CONCAT |
| RELOP | $\Diamond_r$ | ::= | EQ \| NEQ \| GT \| GE \| SGT \| SGE \| LT \| LE \| SLT \| SLE |
| CASTOP | $\Diamond_c$ | ::= | ZeroExt \| SignExt |
| EXPRESSION | ***exp*** | ::= | Num *value size* |
| | | \| | Var *name size* |
| | | \| | PCVar *name size* |
| | | \| | TempVar *name size* |
| | | \| | Name *name* |
| | | \| | UnOp $\Diamond_u$ ***exp*** $\mu$ |
| | | \| | BinOp $\Diamond_b$ ***exp exp*** $\mu$ |
| | | \| | RelOp $\Diamond_r$ ***exp exp*** $\mu$ |
| | | \| | Load $\epsilon$ *size* ***exp*** $\mu$ |
| | | \| | ITE ***exp exp exp*** $\mu$ |
| | | \| | Cast $\Diamond_c$ *size* ***exp*** $\mu$ |
| | | \| | Extract ***exp*** *pos size* |
| | | \| | Undefined *size* |
| STATEMENT | ***stmt*** | ::= | ISMark *addr len* |
| | | \| | IEMark *addr* |
| | | \| | LMark *name* |
| | | \| | Put ***exp exp*** |
| | | \| | Store $\epsilon$ ***exp exp*** |
| | | \| | Jmp ***exp*** |
| | | \| | CJmp ***exp exp exp*** |
| | | \| | InterJmp ***exp exp*** |
| | | \| | InterCJmp ***exp exp exp*** |
| | | \| | SideEffect *SideEffect* |

IR syntax of B2R2

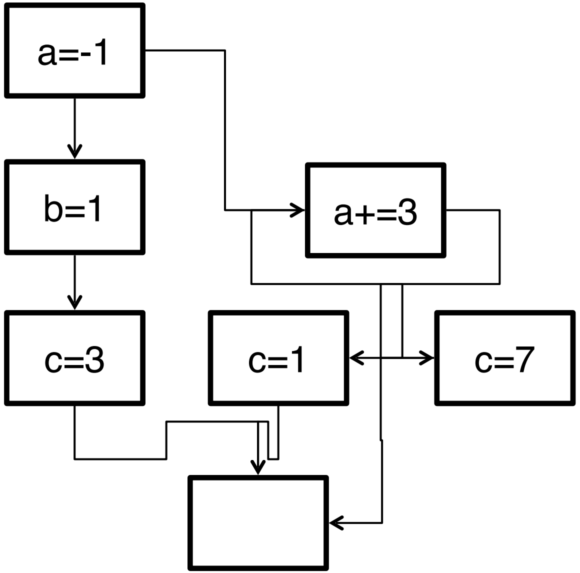
# Data flow analysis

Flow sensitive

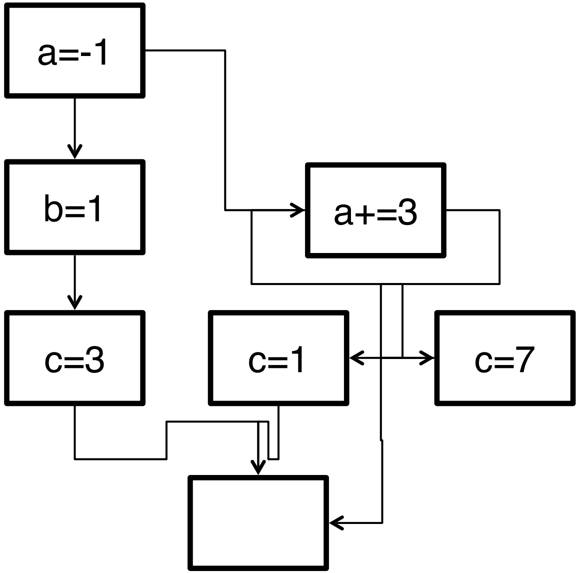
Path sensitive

# Data flow analysis

Flow sensitive

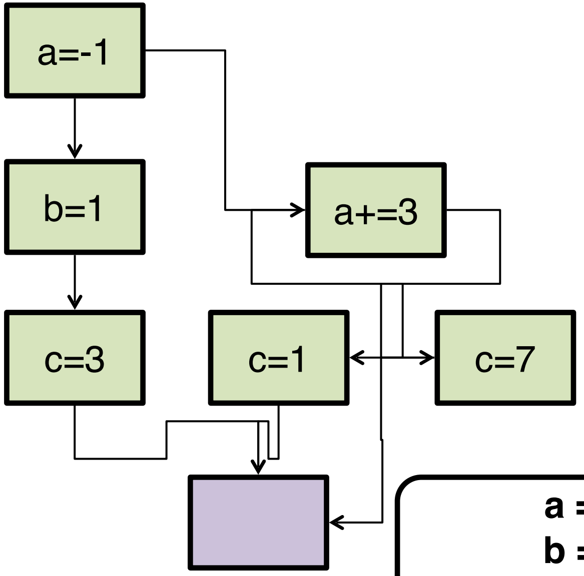



Path sensitive



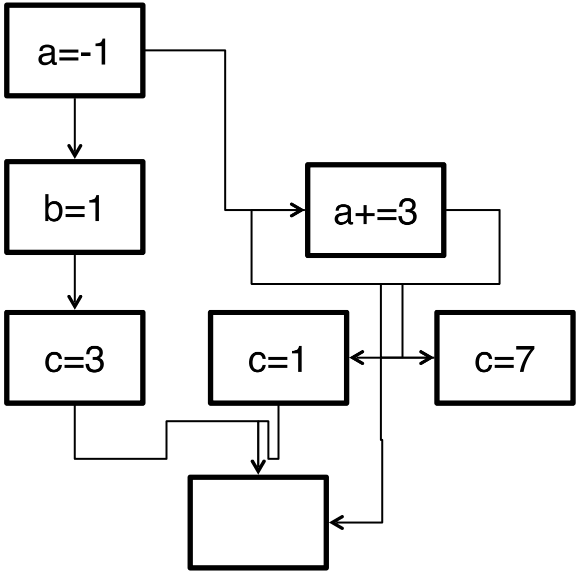
# Data flow analysis

Flow sensitive



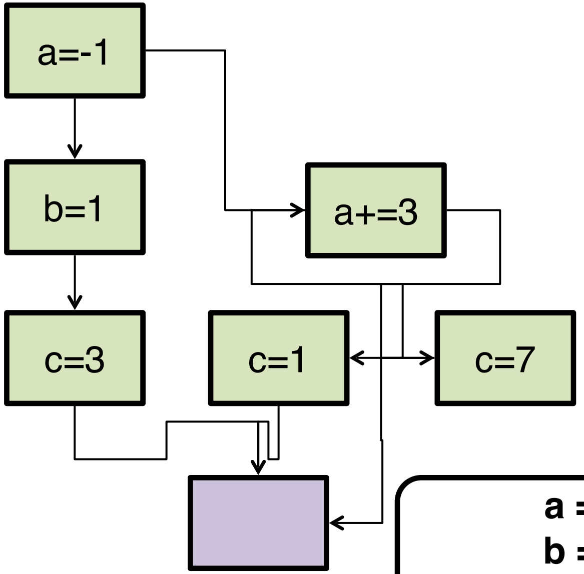

**a = 0[2,2]**
**b = 0[1,1]**
**c = 1[1,7]**

Path sensitive



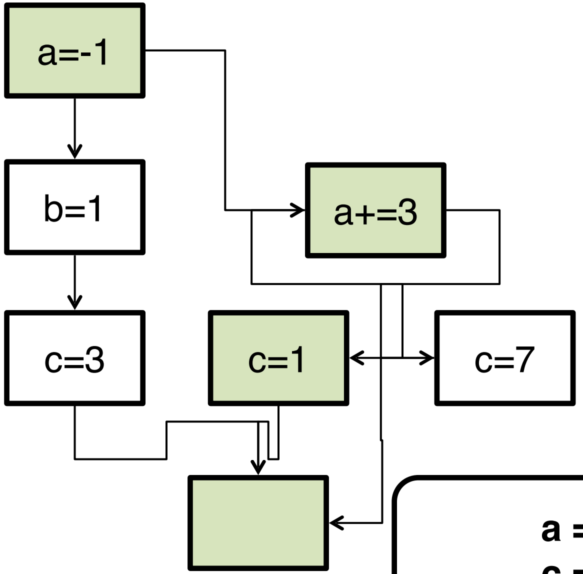
# Data flow analysis

Flow sensitive




**a = 0[2,2]**
**b = 0[1,1]**
**c = 1[1,7]**

Path sensitive

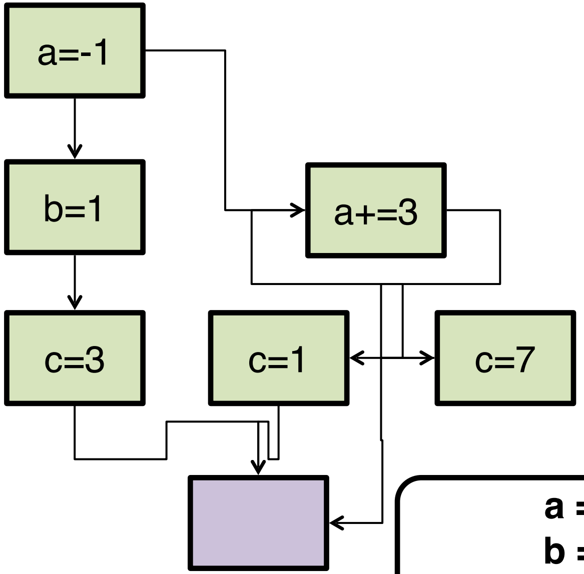



**a = 0[2,2]**
**c = 0[1,1]**

# Data flow analysis

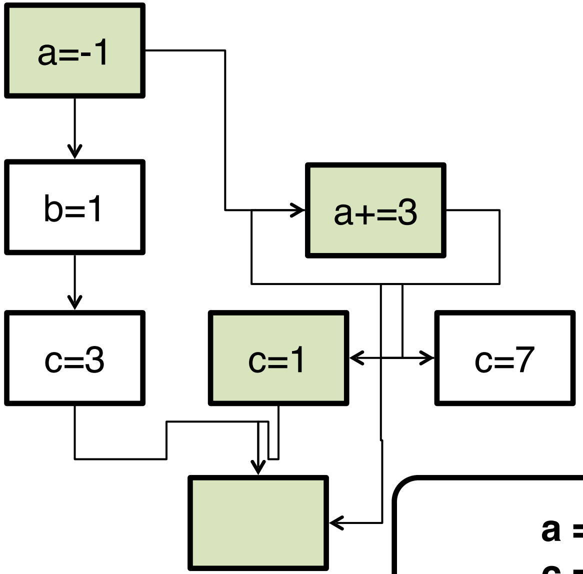
## Flow sensitive

- Recovery rate (vs path sensitive)
  - Analyze all basic blocks




**a = 0[2,2]**
**b = 0[1,1]**
**c = 1[1,7]**

## Path sensitive

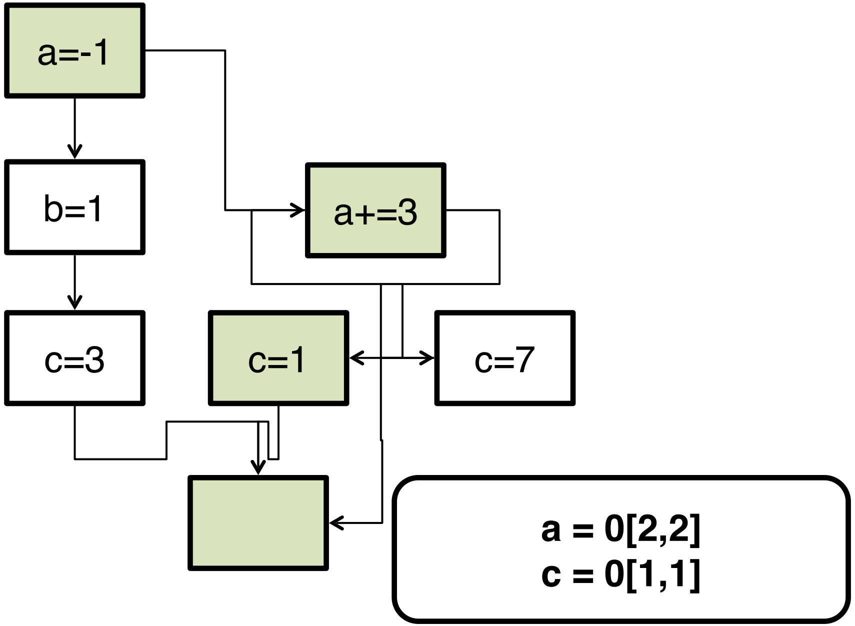



**a = 0[2,2]**
**c = 0[1,1]**

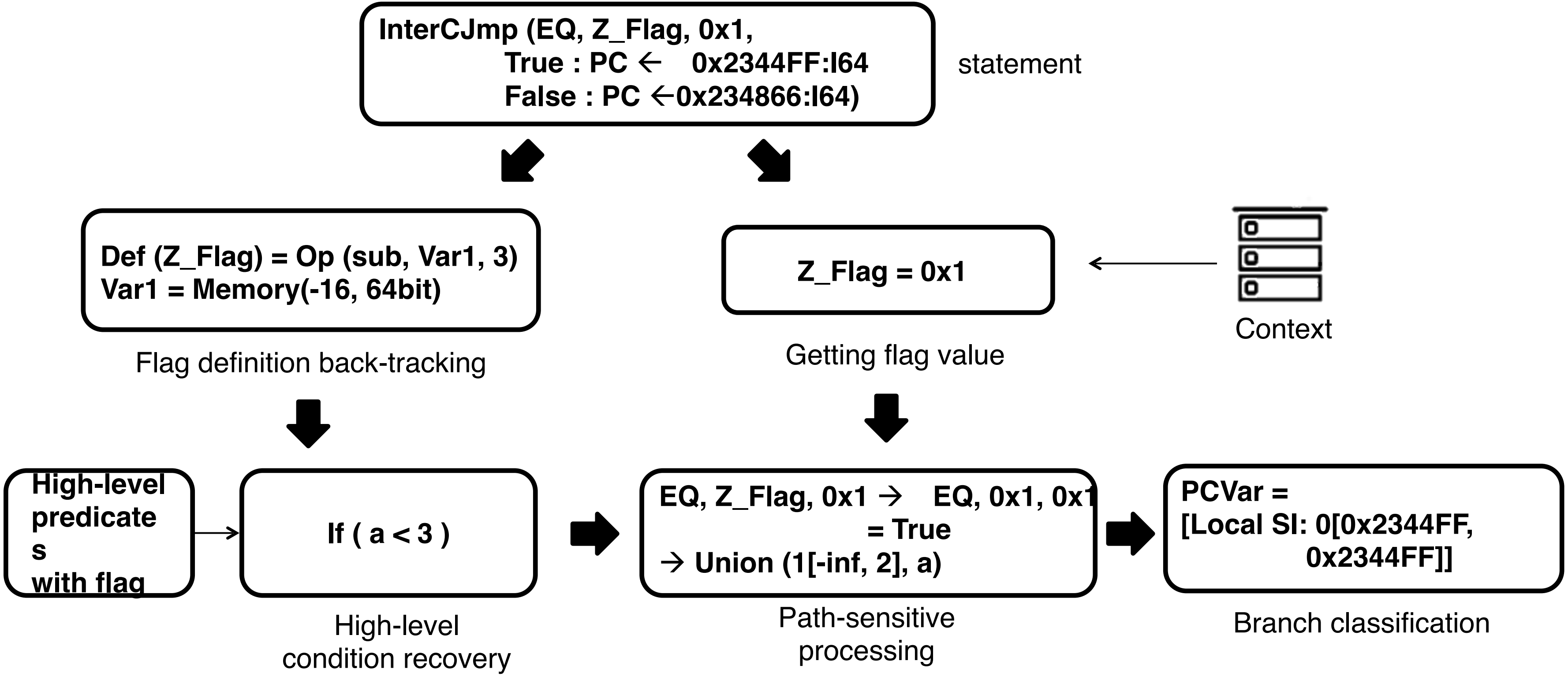
# Data flow analysis

Path sensitive

- Pros
  - Completeness (vs Flow sensitive)
    - Do not merge value sets
  - Simplified value-set structure
    - Do not have multiple memory region at same time
  - Efficiency
    - No analysis a dead code
- Cons
  - Path explosion
    - Unknown value such as user input



# Path-sensitive analysis



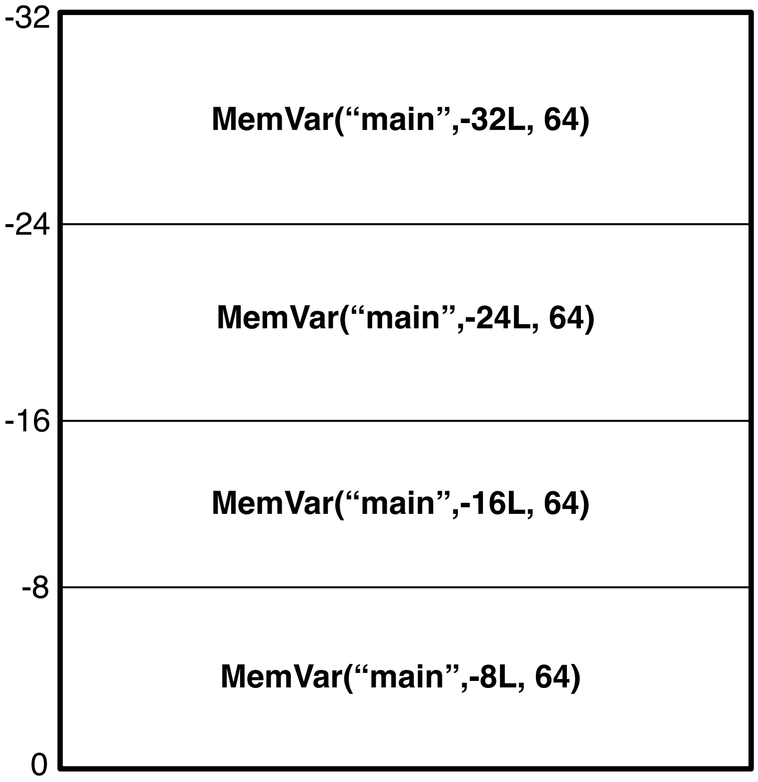
# Memory load & store

- Problem
  - Do not consider overlap memory
    ex) array optimization

# Memory load & store

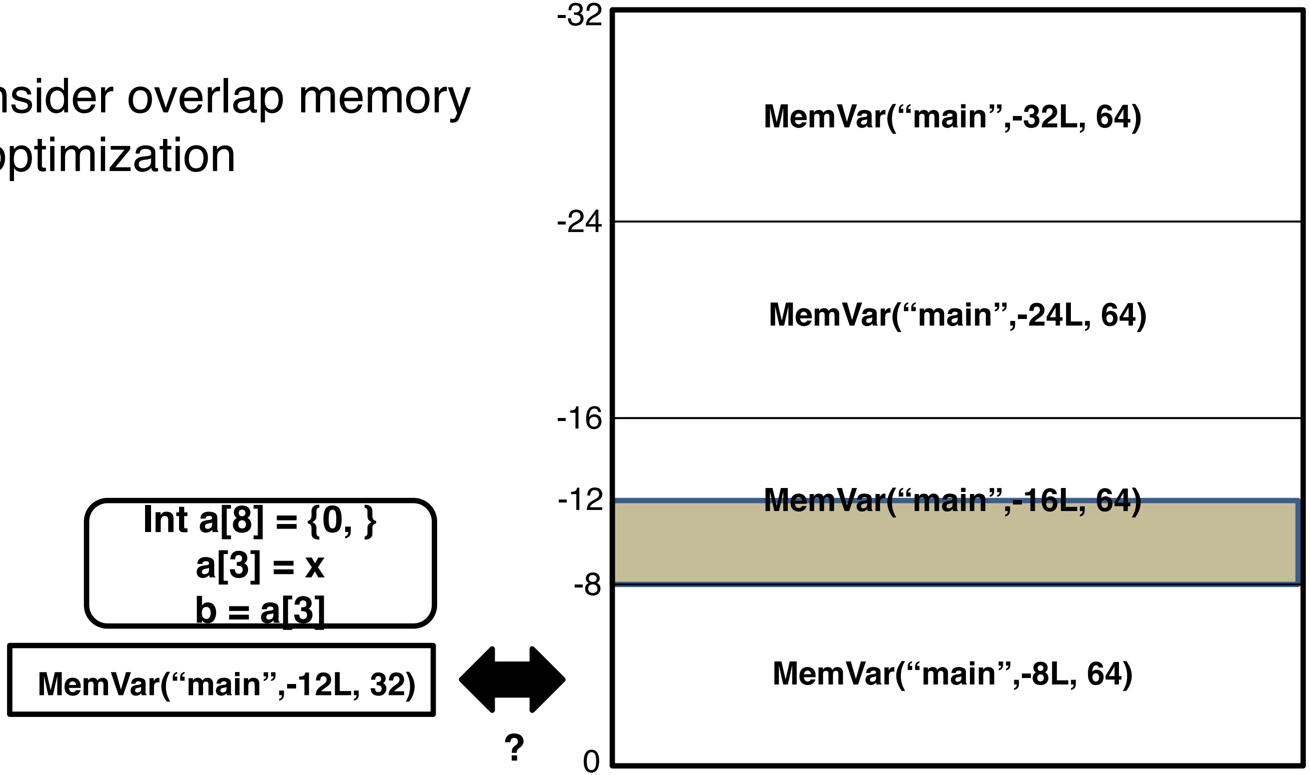
- Problem
  - Do not consider overlap memory
    ex) array optimization

**Int a[8] = {0, }**



# Memory load & store

- Problem
  - Do not consider overlap memory
    ex) array optimization



# Memory load & store

*function* (Memory Load)

Let $vs_{R1} = in[R1],\ \ (F,P) = *\,(vs_{R1}.Si,\ \ vs_{R1}.size)$

if $(|P| = \ 0)$ then

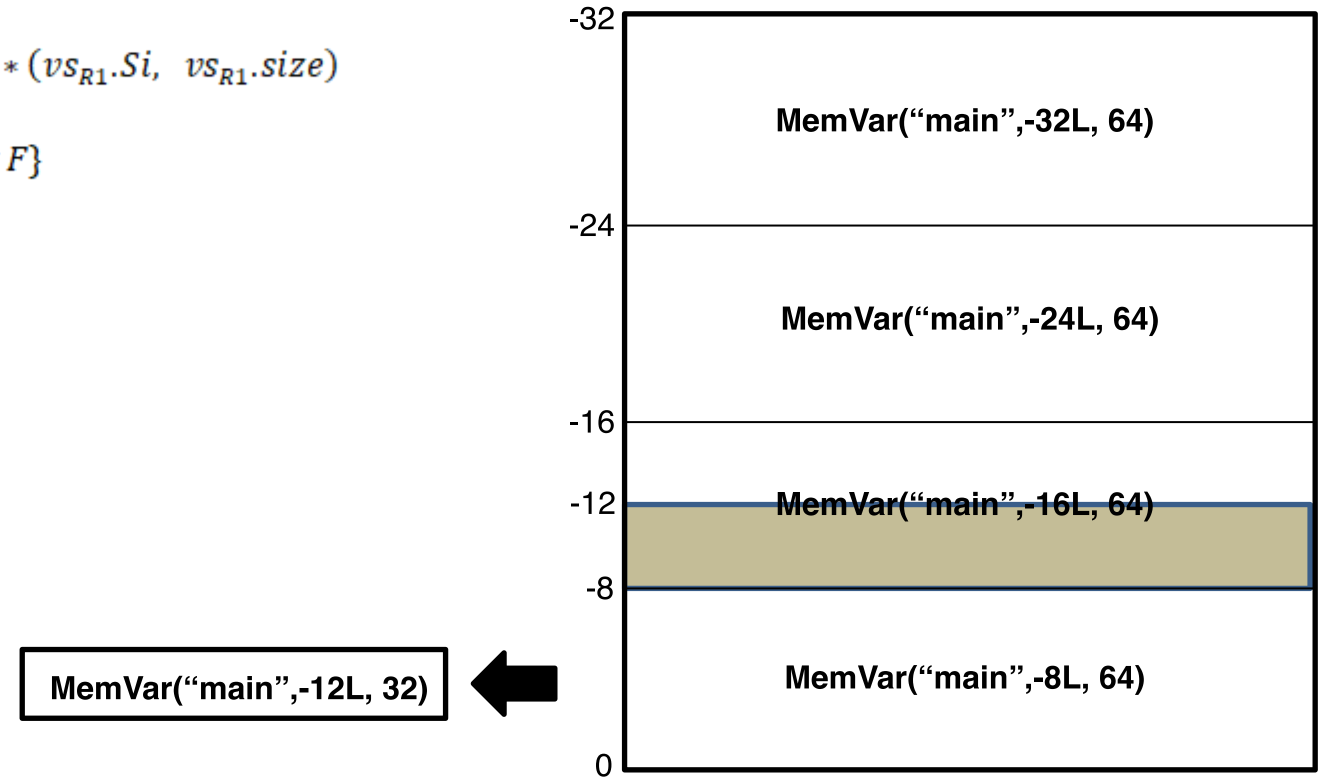
$\quad out[R1] = \cup^{vs}\{in[v] \mid v \in F\}$

else

$\quad out[R1] = T^{vs}$

end if

return out






SysSec System Security Lab

# Memory load & store

*function* (Memory Load)

Let $vs_{R1} = in[R1],\ \ (F, P) = *(vs_{R1}.Si,\ \ vs_{R1}.size)$

if $(|P| = 0)$ then

  $out[R1] = \cup^{vs} \{in[v] \mid v \in F\}$

else

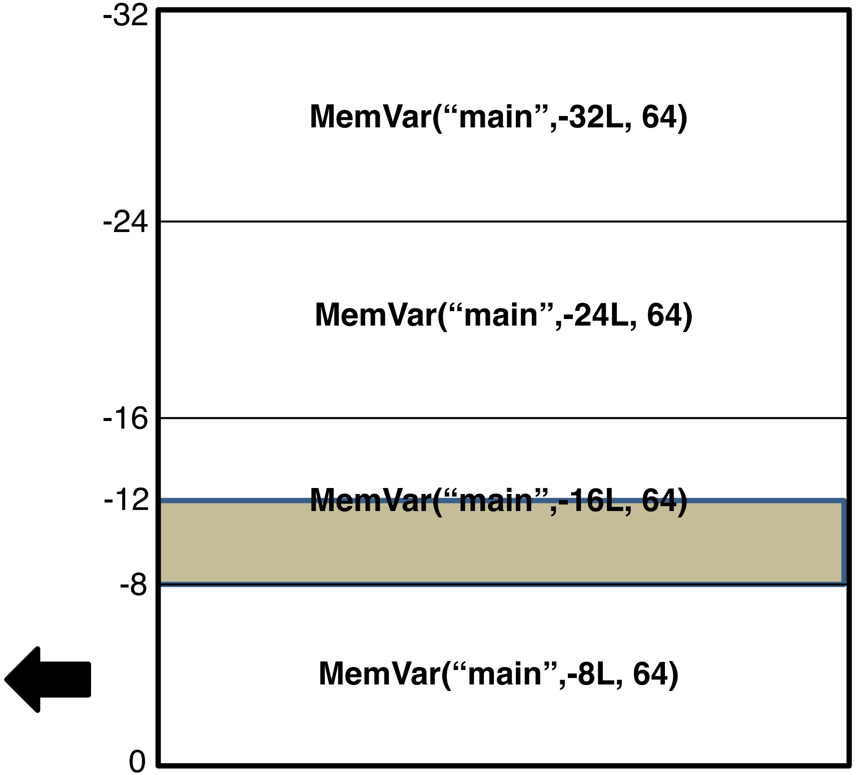
  $out[R1] = T^{vs}$

end if

return out

Do not infer overlapped memory
(Over-approximation)

MemVar(“main”,-12L, 32)






SysSec
System Security Lab

# Memory load & store

$function$ (Memory Load)
Let $vs_{R1} = in[R1], \ (F, P) = *(vs_{R1}.Si, \ vs_{R1}.size)$
if $(|P| = 0)$ then
$\quad out[R1] = \cup^{vs} \{in[v] \mid v \in F\}$
else
$\quad out[R1] = T^{vs}$
end if
return out

Do not infer overlapped memory
(Over-approximation)

$function$ (Memory Load)
Let $vs_{R1} = in[R1], \ (F, P) = *(vs_{R1}.Si, \ vs_{R1}.size)$
if $(|P| = 0)$ then
$\quad out[R1] = \cup^{vs} \{in[v] \mid v \in F\}$
else if $(|F| = 0 \ and \ |P| > 0)$
$\quad$ for each $v \in P \ or \ v \in F$ do
$\quad$ if $(vs_{t1} > v)$
$\quad\quad$ Let $vs_{t1} = in[v] \gg^{vs} c$
$\quad\quad out[R1] = \cup^{vs} vs_{t1}$
$\quad$ ...
$\quad$ end for
else
$\quad out[R1] = T^{vs}$
end if
return out

# Memory load & store

*function* (Memory Store)

Let $vs_{R1} = in[R1],\ vs_{R2} = in[R2],\ (F, P) = *\,(vs_{R1}.Si,\ \ vs_{R1}.size)$

if $(|F| = 1\ \wedge |P| = 0)$ then

  $out[v] = vs_{R2}$, where $v\ \in F$

else

  …

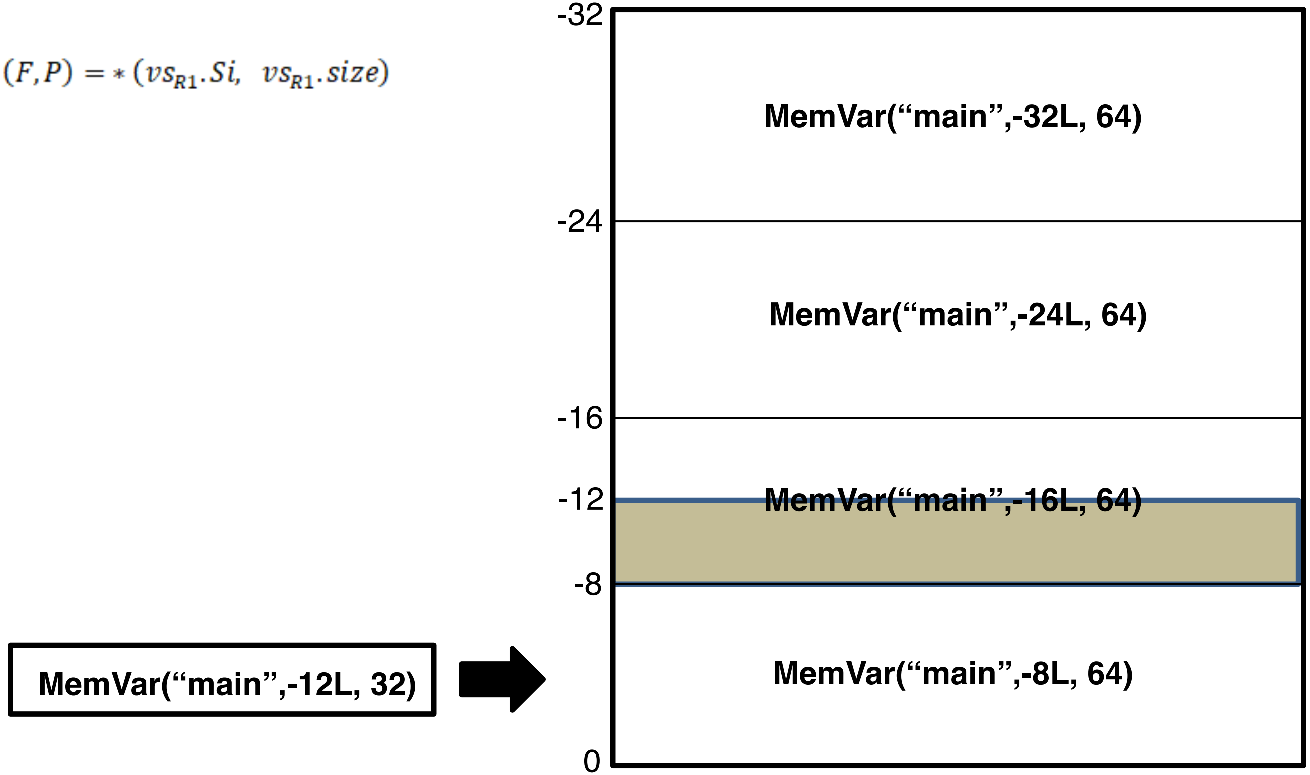
end if

for each $v\ \in P$ do

  $out[v] = T^{vs}$

end for

return out



# Memory load & store

*function* (Memory Store)

Let $vs_{R1} = in[R1],\ vs_{R2} = in[R2],\ (F, P) = *\,(vs_{R1}.Si,\ \ vs_{R1}.size)$

if $(|F| = 1 \wedge |P| = 0)$ then

  $out[v] = vs_{R2}$, where $v \in F$

else

  ...

end if

for each $v \in P$ do

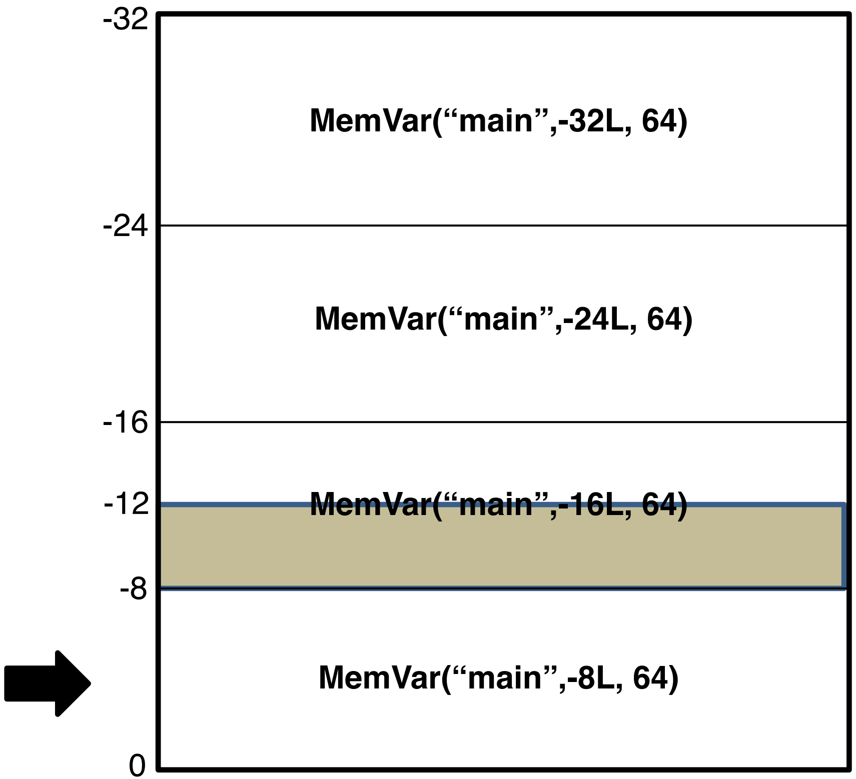
  $out[v] = T^{vs}$

end for

return out

Do not infer overlapped memory (Over-approximation)

**MemVar(“main”,-12L, 32)**



# Memory load & store

*function* (Memory Store)
Let $vs_{R1} = in[R1],\ vs_{R2} = in[R2],\ (F, P) = *\,(vs_{R1}.Si,\ \ vs_{R1}.size)$
if $(|F| = 1\ \wedge |P| = 0)$ then
 $out[v] = vs_{R2}$, where $v\ \in F$
else
 ...
end if
for each $v\ \in P$ do
 $out[v] = T^{vs}$
end for
return out

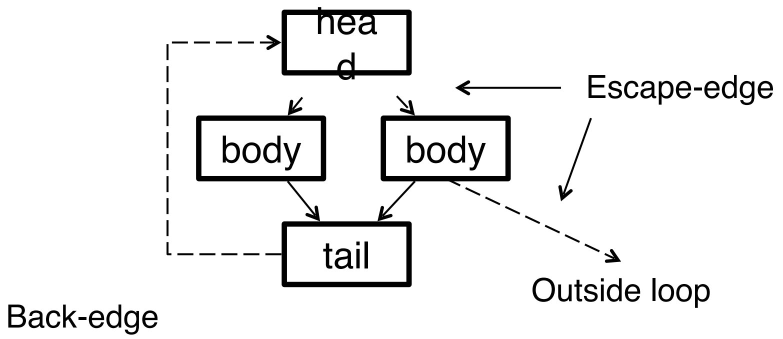
Do not infer overlapped memory
(Over-approximation)

*function* (Memory Store)
Let $vs_{R1} = in[R1],\ vs_{R2} = in[R2],\ (F, P) = *\,(vs_{R1}.Si,\ \ vs_{R1}.size)$
if $(|F| = 1\ \wedge |P| = 0)$ then
 $out[v] = vs_{R2}$, where $v\ \in F$
else
 ...
end if
for each $v\ \in P\ or\ v\ \in F$ do
 if $(vs_{R2}\ >\ v)$
  $out[v] =\ vs_{R2}\ \ll^{vs}\ c$
 ...
end for
return out

# Loop Analysis

- Loop pre-detection process
  1. Find back-edge, tail, and head with CFG
  2. Back-trace from tail to head → loop body
  3. Save edges to escape loop from head
     - If a loop has multiple conditions,
       we must pass multiple edges to escape
  4. Merge a loop if tail or head already exist

- Loop escape
  - If current BBL is not in loop body
  - Current state = previous state
  - Loop count > x (for infinite loop)



# Evaluation process

- Goal
  - Show variables recovery rate
    - Precision, recall

- How?
  - Get local variables with dwarf information (ground truth)
  - Compare our tool with other tool
  - Compare various options
    - InterProcedural vs interProcedural + intraProcedural
    - O0 vs O3

- Limitation
  - DWARF parsing error
  - Path explosion
    - Loop, implementation error, …

# Experimental setup

- Evaluation dataset
  - GCC 8.2.0
  - Total 89 coreutils-8.29 program
- Ground truth
  - llvm-dwarfdump v6.0.0
- Comparison target
  - IDA Pro v6.95
- Machine spec
  - Intel® Core™ i7-8700 CPU @ 3.20GHz × 12
  - DDR3 RAM 15.6 GB
  - 512GB SSD
  - Ubuntu 18.04

# Evaluation

- vs IDA Pro v6.95
  - Recall
    - Unvisited BBL by path-sensitive
    - Over-approximation in loop and array
  - Precision
    - DWARF parsing error
    - …

| | IR-based VSA | | IDA | |
|---|---|---|---|---|
| | Recall | Precision | Recall | Precision |
| O0-intra | 87% (3.5%) | 55% (4.1%) | 91% (3.2%) | 80% (2.3%) |
| O0-inter | 84% (3.8%) | 56% (4.4%) | 91% (3.1%) | 80% (2.3%) |
| O3-intra | 12% (10.0%) | 4% (3.6%) | 52% (14.0%) | 83% (12.2%) |
| O3-inter | 17% (13.9%) | 4% (3.4%) | 52% (15.0%) | 83% (12.5%) |

# Evaluation

- vs IDA Pro v6.95
  - Recall
    - Unvisited BBL by path-sensitive
    - Over-approximation in loop and array
  - Precision
    - DWARF parsing error
    - …

| | IR-based VSA | | IDA | |
|---|---|---|---|---|
| | Recall | Precision | Recall | Precision |
| O0-intra | 87% (3.5%) | 55% (4.1%) | 91% (3.2%) | 80% (2.3%) |
| O0-inter | 84% (3.8%) | 56% (4.4%) | 91% (3.1%) | 80% (2.3%) |
| O3-intra | 12% (10.0%) | 4% (3.6%) | 52% (14.0%) | 83% (12.2%) |
| O3-inter | 17% (13.9%) | 4% (3.4%) | 52% (15.0%) | 83% (12.5%) |

# Evaluation

❖ vs IDA Pro v6.95

– Recall

  ▪ Unvisited BBL by path-sensitive

  ▪ Over-approximation in loop and array

– Precision

  ▪ DWARF parsing error

  ▪ …

| | IR-based VSA | |
|---|---|---|
| | BBL | Recall |
| High-coverage | 89.5% | 95.7% |
| Low-coverage | 34.3% | 34.8% |

| | IR-based VSA | | IDA | |
|---|---|---|---|---|
| | Recall | Precision | Recall | Precision |
| O0-intra | 87% (3.5%) | 55% (4.1%) | 91% (3.2%) | 80% (2.3%) |
| O0-inter | 84% (3.8%) | 56% (4.4%) | 91% (3.1%) | 80% (2.3%) |
| O3-intra | 12% (10.0%) | 4% (3.6%) | 52% (14.0%) | 83% (12.2%) |
| O3-inter | 17% (13.9%) | 4% (3.4%) | 52% (15.0%) | 83% (12.5%) |

| | IR-based VSA | | IDA |
|---|---|---|---|
| | BBL | BBL (by function) | BBL |
| O0-intra | 77.9% | 79.0% (7.0%) | * |
| O0-inter | 69.0% | 71.0% (9.6%) | * |
| O3-intra | 73.2% | 73.4% (6.6%) | * |
| O3-inter | 71.7% | 74.0% (16.0%) | * |

# Evaluation

- vs IDA Pro v6.95
  - Recall
    - Unvisited BBL by path-sensitive
    - Over-approximation in loop and array
  - Precision
    - DWARF parsing error
    - …

| | IR-based VSA | | IDA | |
|---|---|---|---|---|
| | Recall | Precision | Recall | Precision |
| O0-intra | 87% (3.5%) | 55% (4.1%) | 91% (3.2%) | 80% (2.3%) |
| O0-inter | 84% (3.8%) | 56% (4.4%) | 91% (3.1%) | 80% (2.3%) |
| O3-intra | 12% (10.0%) | 4% (3.6%) | 52% (14.0%) | 83% (12.2%) |
| O3-inter | 17% (13.9%) | 4% (3.4%) | 52% (15.0%) | 83% (12.5%) |

| | IR-based VSA (No-array) | | IDA (No-array) | |
|---|---|---|---|---|
| | Recall | Precision | Recall | Precision |
| O0-intra | 90% (2.7%) | 55% (4.1%) | 96% (1.4%) | 80% (2.3%) |
| O0-inter | 88% (3.6%) | 56% (4.4%) | 96% (1.5%) | 80% (2.3%) |
| O3-intra | 14% (11.6%) | 3% (3.1%) | 57% (13.4%) | 83% (12.2%) |
| O3-inter | 18% (14.6%) | 3% (3.3%) | 58% (15.1%) | 83% (12.5%) |

# Evaluation

- vs IDA Pro v6.95
  - Recall
    - Unvisited BBL by path-sensitive
    - Over-approximation in loop and array
  - Precision
    - DWARF parsing error
    - …

| | IR-based VSA | | IDA | |
|---|---|---|---|---|
| | Recall | Precision | Recall | Precision |
| O0-intra | 87% (3.5%) | 55% (4.1%) | 91% (3.2%) | 80% (2.3%) |
| O0-inter | 84% (3.8%) | 56% (4.4%) | 91% (3.1%) | 80% (2.3%) |
| O3-intra | 12% (10.0%) | 4% (3.6%) | 52% (14.0%) | 83% (12.2%) |
| O3-inter | 17% (13.9%) | 4% (3.4%) | 52% (15.0%) | 83% (12.5%) |

| | IR-based VSA (No-array) | | IDA (No-array) | |
|---|---|---|---|---|
| | Recall | Precision | Recall | Precision |
| O0-intra | 90% (2.7%) | 55% (4.1%) | 96% (1.4%) | 80% (2.3%) |
| O0-inter | 88% (3.6%) | 56% (4.4%) | 96% (1.5%) | 80% (2.3%) |
| O3-intra | 14% (11.6%) | 3% (3.1%) | 57% (13.4%) | 83% (12.2%) |
| O3-inter | 18% (14.6%) | 3% (3.3%) | 58% (15.1%) | 83% (12.5%) |

# Evaluation

- vs IDA Pro v6.95
  - Recall
    - Unvisited BBL by path-sensitive
    - Over-approximation in loop and array
  - Precision
    - DWARF parsing error
    - …

| | IR-based VSA | | IDA | |
|---|---|---|---|---|
| | Recall | Precision | Recall | Precision |
| O0-intra | 87% (3.5%) | 55% (4.1%) | 91% (3.2%) | 80% (2.3%) |
| O0-inter | 84% (3.8%) | 56% (4.4%) | 91% (3.1%) | 80% (2.3%) |
| O3-intra | 12% (10.0%) | 4% (3.6%) | 52% (14.0%) | 83% (12.2%) |
| O3-inter | 17% (13.9%) | 4% (3.4%) | 52% (15.0%) | 83% (12.5%) |

# Evaluation

- vs IDA Pro v6.95
  - Recall
    - Unvisited BBL by path-sensitive
    - Over-approximation in loop and array
  - Precision
    - DWARF parsing error
    - …

| | IR-based VSA | | IDA | |
|---|---|---|---|---|
| | Recall | Precision | Recall | Precision |
| O0-intra | 87% (3.5%) | 55% (4.1%) | 91% (3.2%) | 80% (2.3%) |
| O0-inter | 84% (3.8%) | 56% (4.4%) | 91% (3.1%) | 80% (2.3%) |
| O3-intra | 12% (10.0%) | 4% (3.6%) | 52% (14.0%) | 83% (12.2%) |
| O3-inter | 17% (13.9%) | 4% (3.4%) | 52% (15.0%) | 83% (12.5%) |

```
main
{
    char a[10] = {'0', }
    char b[10] = {'0', }

    for (i=0; i<sizeof(a); i++)
        b[i] = a[i]
}
```

# Evaluation

❖ vs IDA Pro v6.95

- Recall
  - Unvisited BBL by path-sensitive
  - Over-approximation in loop and array
- Precision
  - DWARF parsing error
  - …

| | IR-based VSA | | IDA | |
|---|---|---|---|---|
| | Recall | Precision | Recall | Precision |
| O0-intra | 87% (3.5%) | 55% (4.1%) | 91% (3.2%) | 80% (2.3%) |
| O0-inter | 84% (3.8%) | 56% (4.4%) | 91% (3.1%) | 80% (2.3%) |
| O3-intra | 12% (10.0%) | 4% (3.6%) | 52% (14.0%) | 83% (12.2%) |
| O3-inter | 17% (13.9%) | 4% (3.4%) | 52% (15.0%) | 83% (12.5%) |

```
main
{
    char a[10] = {'0', }
    char b[10] = {'0', }

    for (i=0; i<sizeof(a); i++)
        b[i] = a[i]
}
```

i = [0, +inf]

# Evaluation

- vs IDA Pro v6.95
  - Recall
    - Unvisited BBL by path-sensitive
    - Over-approximation in loop and array
  - Precision
    - DWARF parsing error
    - …

| | IR-based VSA | | IDA | |
|---|---|---|---|---|
| | Recall | Precision | Recall | Precision |
| O0-intra | 87% (3.5%) | 55% (4.1%) | 91% (3.2%) | 80% (2.3%) |
| O0-inter | 84% (3.8%) | 56% (4.4%) | 91% (3.1%) | 80% (2.3%) |
| O3-intra | 12% (10.0%) | 4% (3.6%) | 52% (14.0%) | 83% (12.2%) |
| O3-inter | 17% (13.9%) | 4% (3.4%) | 52% (15.0%) | 83% (12.5%) |

ESP= -4 or -8 or -12 or -13
→ ESP = 1[-13, -4]

Load (offset = ESP, size = 32bit)
→ (-13, 32bit), (-12, 32bit), …, (-4, 32bit)

# Evaluation

- vs IDA Pro v6.95
  - Recall
    - Unvisited BBL by path-sensitive
    - Over-approximation in loop and array
  - Precision
    - DWARF parsing error
    - …

| | IR-based VSA | | IDA | |
|---|---|---|---|---|
| | Recall | Precision | Recall | Precision |
| O0-intra | 87% (3.5%) | 55% (4.1%) | 91% (3.2%) | 80% (2.3%) |
| O0-inter | 84% (3.8%) | 56% (4.4%) | 91% (3.1%) | 80% (2.3%) |
| O3-intra | 12% (10.0%) | 4% (3.6%) | 52% (14.0%) | 83% (12.2%) |
| O3-inter | 17% (13.9%) | 4% (3.4%) | 52% (15.0%) | 83% (12.5%) |

```
0 void func_name (…)
{
    4 int argc (DW_OP_fbreg -212)
    8 char** argv (DW_OP_fbreg -224)
    4 int I (DW_OP_fbreg -52)
    8 char* arg (DW_OP_fbreg -64)
    144 stat st (DW_OP_fbreg -208)
}
```

# Evaluation

- O0 vs O3
  - Recall
    - The number of variables
    - Local variables are saved in registers in O3
  - Precision
    - Stack-related instruction
      - pop, push

| | IR-based VSA | | IDA | |
|---|---|---|---|---|
| | Recall | Precision | Recall | Precision |
| O0-intra | 87% (3.5%) | 55% (4.1%) | 91% (3.2%) | 80% (2.3%) |
| O0-inter | 84% (3.8%) | 56% (4.4%) | 91% (3.1%) | 80% (2.3%) |
| O3-intra | 12% (10.0%) | 4% (3.6%) | 52% (14.0%) | 83% (12.2%) |
| O3-inter | 17% (13.9%) | 4% (3.4%) | 52% (15.0%) | 83% (12.5%) |

# Evaluation

❖ O0 vs O3

– Recall

- The number of variables
- Local variables are saved in registers in O3

– Precision

- Stack-related instruction
  - pop, push

| | IR-based VSA | | IDA | |
|---|---|---|---|---|
| | Recall | Precision | Recall | Precision |
| O0-intra | 87% (3.5%) | 55% (4.1%) | 91% (3.2%) | 80% (2.3%) |
| O0-inter | 84% (3.8%) | 56% (4.4%) | 91% (3.1%) | 80% (2.3%) |
| O3-intra | 12% (10.0%) | 4% (3.6%) | 52% (14.0%) | 83% (12.2%) |
| O3-inter | 17% (13.9%) | 4% (3.4%) | 52% (15.0%) | 83% (12.5%) |

```
; int __cdecl main(int argc, const char **ar
public main
main proc near
argc = rbx                    ; int
argv = rbp                    ; char **
; __unwind {
push    argv
mov     argv, rsi
push    argc
mov     ebx, edi              ; argv0
sub     rsp, 8
mov     rdi, [rsi]
call    set_program_name
```

# Evaluation

- O0 vs O3
  - Recall
    - The number of variables
    - Local variables are saved in registers in O3
  - Precision
    - Stack-related instruction
      - pop, push

| | IR-based VSA | | IDA | |
|---|---|---|---|---|
| | Recall | Precision | Recall | Precision |
| O0-intra | 87% (3.5%) | 55% (4.1%) | 91% (3.2%) | 80% (2.3%) |
| O0-inter | 84% (3.8%) | 56% (4.4%) | 91% (3.1%) | 80% (2.3%) |
| O3-intra | 12% (10.0%) | 4% (3.6%) | 52% (14.0%) | 83% (12.2%) |
| O3-inter | 17% (13.9%) | 4% (3.4%) | 52% (15.0%) | 83% (12.5%) |

```
; int __cdecl main(int argc, const char **ar
public main
main proc near
argc = rbx                  ; int
argv = rbp                  ; char **
; __unwind {
push    argv
mov     argv, rsi
push    argc
mov     ebx, edi            ; argv0
sub     rsp, 8
mov     rdi, [rsi]
call    set_program_name
```

# Discussion & Future work

- Low precision by strided-interval

- Recovery scalability
  - x86, ARM, MIPS, …
  - O1, O2, O3, …
  - Heap

- Path-sensitive
  - Contrary to VSA goal that recover all variables in binary
  - Make path-explosion → branch classification

- Loop analysis
  - Widening fixpoint
  - Afterthought inference
  - …

# Conclusion

- Studied binary analysis, f#, …
- Implemented IR-based VSA
  - Flow-sensitive → Path-sensitive
  - Improve memory handling
- Many future works