# Modular Heap Abstraction-Based Memory Leak Detection for Heap-Manipulating Programs

Longming Dong Ji Wang Liqian Chen

National University of Defense Technology, Changsha, China

05/12/2012 – APSEC 2012

# Outline

- Motivation
- Field-sensitive heap abstraction
- Memory leak detection
- Implementation and experiments
- Conclusion

# Motivation

# Motivation

Dynamic allocated data structures

- examples: lists, trees, etc.
- widely used in practice
  - e.g., operating systems, device drivers, etc.
- **error-prone**
  - memory leak
    - degrade preformance
    - cause memory-intensive or long-time running software to crash
  - dangling reference
  - double free
  - null pointer dereference
  - ...

# Motivation

```
typedef struct list{
    int d;
    struct list* n;
}List;

void f(){
1:   List* x=(List*) malloc(sizeof(List));
2:   x→n =(List*) malloc(sizeof(List));
3:   free(x);
}  Memory leak on x → n
```

Field sensitive analysis of heap manipulating programs

- **problem**: high cost for exact memory layout
- **solution**: abstraction to make the problem tractable
  - proper abstraction according to properties to check
    → simplify the problem & be precise enough

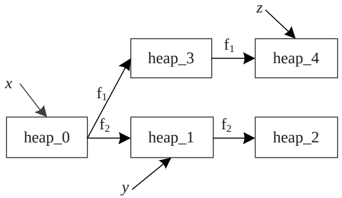
# Field-sensitive heap abstraction

# Concrete heap state

## Shape graph $\langle H, S \rangle$

- the topological structure of heap memory can be described by a directed graph $H = \langle V, E \rangle$ where
  - $V$ denotes the set of heap cells
  - $E : V \xrightarrow{F} V$ denotes the points-to relations between cells via their fields $F$
- $S : PVar \rightarrow V$ where $PVar$ denotes the set of pointer variables

$$PVar = \{x, y, z\}$$
$$V = \{heap\_0, heap\_1, heap\_2, heap\_3, heap\_4\}$$
$$S = \{\langle x, heap\_0\rangle, \langle y, heap\_1\rangle, \langle z, heap\_4\rangle\}$$
$$E = \{heap\_0 \xrightarrow{f_1} heap\_3, heap\_0 \xrightarrow{f_2} heap\_1, heap\_1 \xrightarrow{f_2} heap\_2, heap\_3 \xrightarrow{f_1} heap\_4\}$$




It may cause high memory cost! $\Rightarrow$ abstraction

# Field-sensititive heap abstraction

An abstract domain of member-access distances $\langle D, +, - \rangle$

- elements: a set of abstract distances $D = \{0, 1, 2\}$
  - 0: the current cell **itself** ($p$)
  - 1: member-access with depth **1** ($p \rightarrow f$)
  - 2: member-access with depth **more than 1** ($p \rightarrow f \rightarrow^{\star}$).
- operations: $+, -$ over $D$ (defined in Table 1)

Table 1: Operations over $D$

| $+$ | 0 | 1 | 2 |
|---|---|---|---|
| 0 | {0} | {1} | {2} |
| 1 | {1} | {2} | {2} |
| 2 | {2} | {2} | {2} |

| $-$ | 0 | 1 | 2 |
|---|---|---|---|
| 0 | $\perp$ | $\perp$ | $\perp$ |
| 1 | {1} | {0} | $\perp$ |
| 2 | {2} | {1,2} | $\top$ |

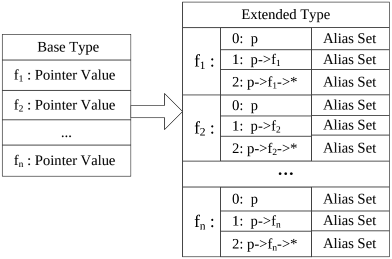
($\perp$: the operator cannot be applied to these operands; $\top$: $\{0, 1, 2\}$)

# Field-sensititive heap abstraction

An extended pointer structure of a pointer $p$

$$\tau_p^\sharp \triangleq \{f_1 : \langle D, 2^{PVar}\rangle;\ f_2 : \langle D, 2^{PVar}\rangle;\ \ldots;\ f_n : \langle D, 2^{PVar}\rangle\}$$

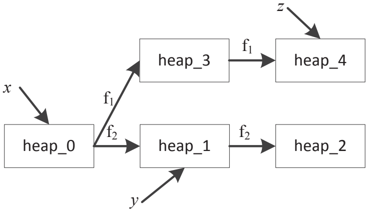
- $f_1, f_2, \ldots, f_n$ denote the $n$ pointer fields of the structure that $p$ points to
- $D = \{0, 1, 2\}$ denotes the set of abstract distances
- $2^{PVar}$ denotes the alias set of accessing field $f_i$ of pointer $p$ with distance $d \in D$



# Field-sensititive heap abstraction

Example:

$$
\begin{aligned}
PVar &= \{x, y, z\} \\
V &= \{heap\_0, heap\_1, heap\_2, heap\_3, heap\_4\} \\
S &= \{\langle x, heap\_0\rangle, \langle y, heap\_1\rangle, \langle z, heap\_4\rangle\} \\
E &= \{heap\_0 \xrightarrow{f_1} heap\_3, heap\_0 \xrightarrow{f_2} heap\_1, \\
&\quad heap\_1 \xrightarrow{f_2} heap\_2, heap\_3 \xrightarrow{f_1} heap\_4\}
\end{aligned}
$$




$$
\begin{aligned}
\tau^\sharp_x &: \{f_1 : \langle 0, \emptyset\rangle, \langle 1, \emptyset\rangle, \langle 2, \{z\}\rangle; \quad f_2 : \langle 0, \emptyset\rangle, \langle 1, \{y\}\rangle, \langle 2, \emptyset\rangle\} \\
\tau^\sharp_y &: \{f_1 : \langle 0, \emptyset\rangle, \langle 1, \perp\rangle; \quad f_2 : \langle 0, \emptyset\rangle, \langle 1, \emptyset\rangle, \langle 2, \perp\rangle\} \\
\tau^\sharp_z &: \{f_1 : \langle 0, \emptyset\rangle, \langle 1, \perp\rangle; \quad f_2 : \langle 0, \emptyset\rangle, \langle 1, \perp\rangle\}
\end{aligned}
$$

# Field-sensititive heap abstraction

## Definition (Abstract Heap State)

The abstract heap state $\mathbb{S}^\sharp$ at each program point of a program *HP* consists of a set of extended structures of all pointer variables:

$$\mathbb{S}^\sharp = \{\tau^\sharp_{p_i} | p_i \in PVar(HP)\}$$

The number of abstract heap states: finite

$$\leq (fn \times 3 \times (2^{pn-1} + 1))^{pn}$$

- $pn$: the number of pointer variables
- $fn$: the maximum number of pointer fields

# Field-sensititive heap abstraction

Alias bit-vector: using **bit-vector** to encode **alias set**

- maintain a variable ordering for all variables in *PVar*
- a alias bit-vector $\overrightarrow{r}^{\sharp}_{x} \in \{0,1\}^{|PVar|}$ is defined as

  $\overrightarrow{r}^{\sharp}_{p}(f_m, d)[i] = 1 \Leftrightarrow v_i$ is an alias of accessing $f_m$ of $p$ with distance $d$

Example: $PVar = \{x, y, z\}$ with variable ordering: $x \prec y \prec z$

$$\overrightarrow{r}^{\sharp}_{x} : \{f_1 : \langle 0, 0\rangle, \langle 1, 0\rangle, \langle 2, 001\rangle;\ f_2 : \langle 0, 0\rangle, \langle 1, 010\rangle, \langle 2, 0\rangle\}$$
$$\overrightarrow{r}^{\sharp}_{y} : \{f_1 : \langle 0, 0\rangle, \langle 1, \perp\rangle;\ f_2 : \langle 0, 0\rangle, \langle 1, 0\rangle, \langle 2, \perp\rangle\}$$
$$\overrightarrow{r}^{\sharp}_{z} : \{f_1 : \langle 0, 0\rangle, \langle 1, \perp\rangle;\ f_2 : \langle 0, 0\rangle, \langle 1, \perp\rangle\}$$

# Field-sensititive heap abstraction

Abstract heap state with a canonical form

## Definition (Saturated abstract state)

An abstract state $\mathbb{S}^\sharp$ is *saturated* iff it satisfies:

1. **Anti-reflexivity**. $\forall p_i. \overrightarrow{r}^\sharp_{p_i}(f_m, 0)[i] = 0$.
2. **Symmetry**. $\forall p_i, p_j.\ \overrightarrow{r}^\sharp_{p_i}(f_m, 0)[j] = 1 \rightarrow \overrightarrow{r}^\sharp_{p_j}(f_m, 0)[i] = 1$.
3. **Transitivity**. $\forall p_i, p_j, p_t.\ \overrightarrow{r}^\sharp_{p_i}(f_m, d_1)[j] = 1 \wedge \overrightarrow{r}^\sharp_{p_j}(f_m, d_2)[t] = 1$ $\rightarrow \overrightarrow{r}^\sharp_{p_i}(f_m, d_1 + d_2)[t] = 1$.

# Memory leak detection

# Syntax of heap-manipulating programs

$p,\ q \in PVar$
$f_1,\ f_2,\ \ldots\ f_i\ \ldots,\ f_m \in Fields$
$AsgnStmt := \ p = null |\ p \rightarrow f_i = null |\ p = q |$
$\qquad p = q \rightarrow f_i |\ p \rightarrow f_i = q |\ p = malloc() |\ p = free()$
$SwitchStmt := \ switch\ e\ \{c_1 : n_1,\ \ldots,\ c_j : c_j,\ \ldots,\ c_k : n_k,\ \ldots\}$
$CallStmt := \ e = g(e_1,\ \ldots,\ e_k)$
$ReturnStmt := \ return\ \ e$
$Stmt := \ AsgnStmt |\ SwitchStmt |\ CallStmt |\ ReturnStmt$
$SequenceStmt := \ Stmt; Stmt$

# Abstract semantics

1. $$[[p_u = null]](\mathbb{S}^\sharp) = \begin{cases} \{\mathbb{S}^\sharp[\overrightarrow{r}^{\sharp}_{p_v}(f_m,d)[u] \leftarrow 0, \overrightarrow{r}^{\sharp}_{p_u}(f_m,d) \leftarrow \bot] \mid v \neq u\} & \text{if } \exists w \neq u \wedge l \in D.\ \overrightarrow{r}^{\sharp}_{p_w}(f_m,l)[u] \neq 0 \vee \overrightarrow{r}^{\sharp}_{p_u}(f_m,0) = \bot \\ \{\textbf{memory\_leak}\} & \text{otherwise} \end{cases}$$

2. $$[[p_u \rightarrow f_i = null]](\mathbb{S}^\sharp) = \begin{cases} \{\mathbb{S}^\sharp[\overrightarrow{r}^{\sharp}_{p_v}(f_m,2) \leftarrow \overrightarrow{r}^{\sharp}_{p_v}(f_m,2) \overrightarrow{-} \overrightarrow{r}^{\sharp}_{p_u}(f_i,l) \overrightarrow{+} \overrightarrow{r}^{\sharp}_{p_u}(f_j,l) \mid v \in \{v \mid \overrightarrow{r}^{\sharp}_{p_v}(f_m,1)[u] = 1 \vee \overrightarrow{r}^{\sharp}_{p_v}(f_m,2)[u] = 1\} \wedge l \in \{1,2\} \wedge f_j \in Fields - \{f_i\}, \overrightarrow{r}^{\sharp}_{p_u}(f_i,l) \leftarrow \bot \mid l \in \{1,2\}, \overrightarrow{r}^{\sharp}_{p_w}(f_i,l) \leftarrow \bot \mid l \in \{1,2\} \wedge w \in \{w \mid \overrightarrow{r}^{\sharp}_{p_u}(f_m,0)[w] = 1\}]\} & \text{if } \overrightarrow{r}^{\sharp}_{p_u}(f_i,1) \neq 0 \\ \{\textbf{memory\_leak}\} & \text{otherwise} \end{cases}$$

3. $$[[p_u = p_v]](\mathbb{S}^\sharp) = \begin{cases} \{\mathbb{S}^\sharp_1[\overrightarrow{r}^{\sharp}_{p_u}(f_m,d) \leftarrow \overrightarrow{r}^{\sharp}_{p_v}(f_m,d)] \mid \mathbb{S}^\sharp_1 \in [[p_u = null]]\mathbb{S}^\sharp\} & \text{if } \exists w \neq u \wedge l \in D. \overrightarrow{r}^{\sharp}_{p_w}(f_m,l)[u] \neq 0 \vee \overrightarrow{r}^{\sharp}_{p_u}(f_m,0) = \bot \\ \{\textbf{memory\_leak}\} & \text{otherwise} \end{cases}$$

4. $$[[p_u = p_v \rightarrow f_i]](\mathbb{S}^\sharp) = \begin{cases} \{\mathbb{S}^\sharp_1[\overrightarrow{r}^{\sharp}_{p_u}(f_m,0) \leftarrow \overrightarrow{r}^{\sharp}_{p_v}(f_m,1), \overrightarrow{r}^{\sharp}_{p_v}(f_m,1)[u] \leftarrow 1] \mid \mathbb{S}^\sharp_1 \in [[p_u = null]]\mathbb{S}^\sharp\} & \text{if } \exists w \neq u \wedge l \in D. \overrightarrow{r}^{\sharp}_{p_w}(f_m,l)[u] \neq 0 \vee \overrightarrow{r}^{\sharp}_{p_u}(f_m,0) = \bot \\ \{\textbf{memory\_leak}\} & \text{otherwise} \end{cases}$$

($\overrightarrow{+}$: bitwise addition; $\overrightarrow{-}$: bitwise subtraction)

# Abstract semantics

5. $[[p_u \rightarrow f_i = p_v]](\mathbb{S}^\sharp) = \{\mathbb{S}_1^\sharp[(\overrightarrow{r}^\sharp_{p_w}(f_i, 1) \leftarrow \overrightarrow{r}^\sharp_{p_v}(f_m, 0))[v] \leftarrow 1, \overrightarrow{r}^\sharp_{p_w}(f_m, 2) \leftarrow \overrightarrow{r}^\sharp_{p_v}(f_m, 1) \overrightarrow{+} \overrightarrow{r}^\sharp_{p_v}(f_m, 2), (\overrightarrow{r}^\sharp_{p_t}(f_m, 2) \leftarrow \overrightarrow{r}^\sharp_{p_t}(f_m, 2) \overrightarrow{+} \overrightarrow{r}^\sharp_{p_v}(f_m, 0) \overrightarrow{+} \overrightarrow{r}^\sharp_{p_v}(f_m, 1) \overrightarrow{+} \overrightarrow{r}^\sharp_{p_v}(f_m, 2))[v] \leftarrow 1] \mid t \in \{t \mid \overrightarrow{r}^\sharp_{p_t}(f_m, 1)[u] = 1 \vee \overrightarrow{r}^\sharp_{p_t}(f_m, 2)[u] = 1\} \wedge w \in \{w \mid \overrightarrow{r}^\sharp_{p_u}(f_m, 0)[w] = 1\} \cup \{u\} \wedge \mathbb{S}_1^\sharp \in [[p_u \rightarrow f_i = null]](\mathbb{S}^\sharp)\}$
   if $\overrightarrow{r}^\sharp_{p_u}(f_i, 1) \neq 0$
   $\{$**memory_leak**$\}$ otherwise

6. $[[p_u = malloc]](\mathbb{S}^\sharp) = \{\mathbb{S}_1^\sharp[\overrightarrow{r}^\sharp_{p_u}(f_m, d) \leftarrow 0] \mid \mathbb{S}_1^\sharp \in [[p_u = null]]\mathbb{S}^\sharp\}$
   if $\exists w \neq u \wedge l \in D.\ \overrightarrow{r}^\sharp_{p_w}(f_m, l)[u] \neq 0 \vee \overrightarrow{r}^\sharp_{p_u}(f_m, 0) = \bot$
   $\{$**memory_leak**$\}$ otherwise

7. $[[p_u = free()]](\mathbb{S}^\sharp) = \{\mathbb{S}^\sharp[\overrightarrow{r}^\sharp_{p_w}(f_m, d)[t] \leftarrow 0, \overrightarrow{r}^\sharp_{p_w}(f_m, d)[u] \leftarrow 0, \overrightarrow{r}^\sharp_{p_t}(f_m, d) \leftarrow \bot, \overrightarrow{r}^\sharp_{p_u}(f_m, d) \leftarrow \bot] \mid t \in \{t \mid \overrightarrow{r}^\sharp_{p_u}(f_m, 0)[t] = 1\} \wedge w \in \{w \mid \overrightarrow{r}^\sharp_{p_u}(f_m, 0)[w] = 0 \wedge w \neq u\}\}$
   if $\overrightarrow{r}^\sharp_{p_u}(f_m, 1) \neq 0 \vee \overrightarrow{r}^\sharp_{p_u}(f_m, 1) = \bot$
   $\{$**memory_leak**$\}$ otherwise

($\overrightarrow{+}$: bitwise addition; $\overrightarrow{-}$: bitwise subtraction)

# Memory leak detection

## Detecting memory leaks in assignments

(a) check whether there are other pointers that can access the cell pointed to by the current pointer $(p_u)$, such as **Rule** 1, 3, 4, 6;

(b) check whether there are other pointers that points to the cell referenced by the pointer field of the current pointer $(p_u \rightarrow f_i)$, such as **Rule** 2, 5;

(c) check whether all pointer fields of the current pointer $(p_u \rightarrow f_i)$ are *null* or pointed to by other pointers, like **Rule** 7.

## Example

1. $$[[p_u = null]](\mathbb{S}^\sharp) = \begin{cases} \{\mathbb{S}^\sharp[\overrightarrow{r}^\sharp_{p_v}(f_m, d)[u] \leftarrow 0, \overrightarrow{r}^\sharp_{p_u}(f_m, d) \leftarrow \bot] | v \neq u\} & \text{if} \quad \exists w \neq u \wedge l \in D.\ \overrightarrow{r}^\sharp_{p_w}(f_m, l)[u] \neq 0 \vee \overrightarrow{r}^\sharp_{p_u}(f_m, 0) = \bot \\ \{\textbf{memory\_leak}\} & \text{otherwise} \end{cases}$$

# Interprocedural memory leak detection

## Big-step abstract semantics

$\mathbb{S}^\sharp_{Of} = [[f(p_0, p_1, ..., p_{k-1})]](\mathbb{S}^\sharp_{If})$, wherein $\mathbb{S}^\sharp_{Of}$ is the postcondition after the running of the callee $f$ under the precondition $\mathbb{S}^\sharp_{If}$.

Procedural summary: $\langle \mathbb{S}^\sharp_{If}, \mathbb{S}^\sharp_{Of} \rangle$

- $\mathbb{S}^\sharp_{If}$: abstract heap state over arguments and global pointers
- $\mathbb{S}^\sharp_{Of}$: abstract heap state over return variable and global pointers

# Example

```
typedef struct list{
   int d;
   struct list* n;
}List;

List* l;

List* f1(List* p){
   if(p!=NULL){
      l=p;
   }
   p=(List*) malloc(sizeof(List));
   p->n=(List*)malloc(sizeof(List));
   return p;
}

void g1(){
   l=NULL;
   List* x=(List*)malloc(sizeof(List));
   List* y=f1(x);
   free(y);
   free(l);
}

void g2(){
   l=NULL;
   List* x=NULL;
   List* z=f1(x);
   free(z);
}
```

Table 2: procedural summary for $f1$

| Precondition | Postcondition |
|---|---|
| $\vec{r}^{\sharp}_{p} : \{n : \langle 0, \bot\rangle\}$<br>$\vec{r}^{\sharp}_{l} : \{n : \langle 0, \bot\rangle\}$ | $\vec{r}^{\sharp}_{ret_{f1}} : \{n : \langle 0, \emptyset\rangle, \langle 1, \emptyset\rangle, \langle 2, \bot\rangle\}$<br>$\vec{r}^{\sharp}_{l} : \{n : \langle 0, \bot\rangle\}$ |
| $\vec{r}^{\sharp}_{p} : \{n : \langle 0, \emptyset\rangle, \langle 1, \bot\rangle\}$<br>$\vec{r}^{\sharp}_{l} : \{n : \langle 0, \bot\rangle\}$ | $\vec{r}^{\sharp}_{ret_{f1}} : \{n : \langle 0, \emptyset\rangle, \langle 1, \emptyset\rangle, \langle 2, \bot\rangle\}$<br>$\vec{r}^{\sharp}_{l} : \{n : \langle 0, \emptyset\rangle, \langle 1, \bot\rangle\}$ |
| $\vec{r}^{\sharp}_{p} : \{n : \langle 0, \emptyset\rangle, \langle 1, \emptyset\rangle, \langle 2, \bot\rangle\}$<br>$\vec{r}^{\sharp}_{l} : \{n : \langle 0, \bot\rangle\}$ | $\vec{r}_{ret_{f1}} : \{n : \langle 0, \emptyset\rangle, \langle 1, \emptyset\rangle, \langle 2, \bot\rangle\}$<br>$\vec{r}^{\sharp}_{l} : \{n : \langle 0, \emptyset\rangle, \langle 1, \emptyset\rangle, \langle 2, \bot\rangle\}$ |
| $\vec{r}^{\sharp}_{p} : \{n : \langle 0, \emptyset\rangle, \langle 1, \emptyset\rangle, \langle 2, \emptyset\rangle\}$<br>$\vec{r}^{\sharp}_{l} : \{n : \langle 0, \bot\rangle\}$ | $\vec{r}^{\sharp}_{ret_{f1}} : \{n : \langle 0, \emptyset\rangle, \langle 1, \emptyset\rangle, \langle 2, \bot\rangle\}$<br>$\vec{r}^{\sharp}_{l} : \{n : \langle 0, \emptyset\rangle, \langle 1, \emptyset\rangle, \langle 2, \emptyset\rangle\}$ |

Fixpoint iteration algorithm for analysis

- to compute the abstract heap state for each program point
- worklist-based
- always terminate (without need of widening)
  - maximum number of heap abstract states: $(fn \times 3 \times (2^{pn-1} + 1))^{pn}$

# Implementation and experiments

# Prototype

## Heapcheck

- a field and context sensitive interprocedural memory leak detector
- based on Crystal (a program analysis system for C) [1]
- preprocessing process
  - slicing
  - transforming the input program into a SSA-like form by instrumenting new pointer variables

| Pointer assignments | SSA-like assignments |
|---|---|
| $p \to f_i = q \to f_j$ | $pt_0 = q \to f_j;\ p \to f_i = pt_0$ |
| $p = p \to f_i$ | $pt_1 = p \to f_i;\ p = pt_1$ |
| $p \to f_i = malloc$ | $pt_2 = malloc;\ p \to f_i = pt_2$ |
| $p = q \to f_i \to f_j$ | $pt_3 = q \to f_i;\ p = pt_3 \to f_j$ |
| $p \to f_i = free()$ | $pt_4 = p \to f_i;\ pt_4 = free()$ |

[1] https://www.cs.cornell.edu/projects/crystal/

# Experiments

## Results on benchmark programs (memory leak)

| Programs | Size (Kloc) | Preprocess time (Sec) | Analysis time (Sec) | | Memory (MB) | | Reported alarms |
|---|---|---|---|---|---|---|---|
| | | | Without sum. | Sum.-based | Without sum. | Sum.-based | (#*fp*/#*total*) |
| 164.gzip | 7.7 | 1.19 | 0.31 | 0.33 | 27 | 6 | 0 |
| 175.vpr | 17 | 1.84 | 2.83 | 1.11 | 194 | 86.7 | 1/1 |
| 179.art | 1.2 | 0.32 | 0.1 | 0.1 | 34.4 | 33 | 0 |
| 186.crafty | 21.7 | 3.13 | 7.56 | 6.98 | 295 | 258 | 0 |
| 188.ammp | 13.2 | 1.88 | 1.22 | 0.21 | 135 | 60.2 | 0 |
| 300.twolf | 19.9 | 3.05 | 7.38 | 4.31 | 442 | 195 | 0/3 |
| 176.gcc | 210 | 8.35 | 106.62 | 61.04 | 4596 | 920 | 2/17 |
| tar-1.12 | 11.7 | 1.08 | 18.98 | 9.09 | 239 | 178 | 0/5 |
| openssh | 58.3 | 20.55 | 2.61 | 1.44 | 186 | 144.3 | 2/14 |
| openssl | 36 | 8.47 | 0.46 | 0.44 | 73.5 | 40.7 | 6/11 |

- Real bugs found (#*totoal* − #*fp*)
  1. ignoring judging whether all the **sub-level pointer fields** are null when deallocating the heap cell pointed to by a pointer
  2. the heap cell pointed to by a **local** pointer variable is not deallocated at the return site

# Experiments

## Results on benchmark programs (memory leak)

| Programs | Size (Kloc) | Preprocess time (Sec) | Analysis time (Sec) | | Memory (MB) | | Reported alarms |
|---|---|---|---|---|---|---|---|
| | | | Without sum. | Sum.-based | Without sum. | Sum.-based | (#*fp*/#*total*) |
| 164.gzip | 7.7 | 1.19 | 0.31 | 0.33 | 27 | 6 | 0 |
| 175.vpr | 17 | 1.84 | 2.83 | 1.11 | 194 | 86.7 | 1/1 |
| 179.art | 1.2 | 0.32 | 0.1 | 0.1 | 34.4 | 33 | 0 |
| 186.crafty | 21.7 | 3.13 | 7.56 | 6.98 | 295 | 258 | 0 |
| 188.ammp | 13.2 | 1.88 | 1.22 | 0.21 | 135 | 60.2 | 0 |
| 300.twolf | 19.9 | 3.05 | 7.38 | 4.31 | 442 | 195 | 0/3 |
| 176.gcc | 210 | 8.35 | 106.62 | 61.04 | 4596 | 920 | 2/17 |
| tar-1.12 | 11.7 | 1.08 | 18.98 | 9.09 | 239 | 178 | 0/5 |
| openssh | 58.3 | 20.55 | 2.61 | 1.44 | 186 | 144.3 | 2/14 |
| openssl | 36 | 8.47 | 0.46 | 0.44 | 73.5 | 40.7 | 6/11 |

- Precision
  - false positive rate: about 32% on openssh and openssl
    - compared with [B. Hackett, R. Rugina. Region-based shape analysis with tracked locations. POPL05]: about 64% (openssh: 16/26; openssh: 9/13)

# Conclusion

# Conclusion

## Summary

- a field sensitive heap abstraction based on **member-access distances** and **alias bit-vector** domain
- a field and context sensitive interprocedural memory leak detection algorithm based on **summaries**
- experimental evaluations
  - our approach is scalable with satisfied precision in detecting **memory leaks** for large heap-manipulating programs

THANK YOU!

QUESTIONS?