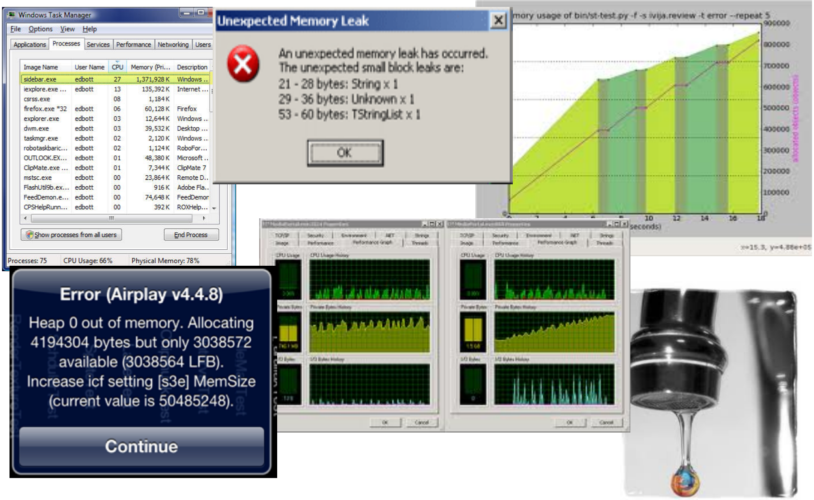
Static Memory Leak Detection Using Full-Sparse Value-Flow Analysis

Yulei Sui, Ding Ye, Jingling Xue

School of Computer Science and Engineering
University of New South Wales
2052 Australia

July 19, 2012

Memory Leaks: Performance and Reliability Issues!



@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group
UNSW THE UNIVERSITY OF NEW SOUTH WALES

What is a Memory Leak?

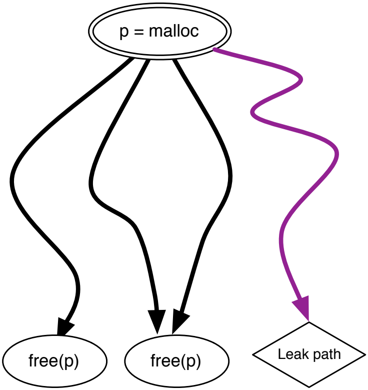
- Gradual loss of available memory for a program
- C/C++: a dynamically allocated object is not freed along some execution path of the program

```
/* Samba – libads/ldap.c:ads_leave_realm */
host = strdup(hostname);
if (...) { ...; return ADS_ERROR_SYSTEM(ENOENT); }
```

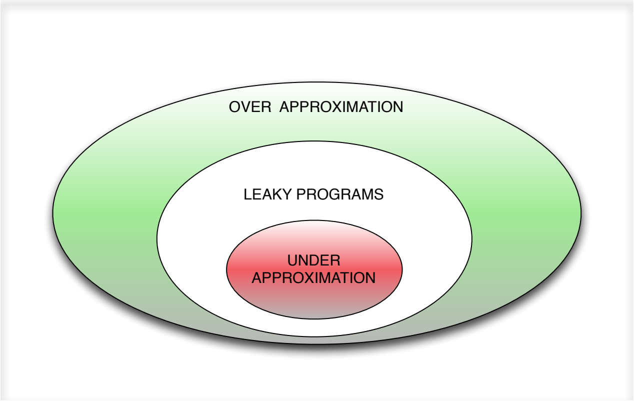
The programmer forgot to free **host** on error.

Static Memory Leak Detection

- Source-Sink Problem: every object created at an allocation site (**a source**) must eventually reach at least one free site (**a sink**) during any program execution path.



Static Memory Leak Detectors

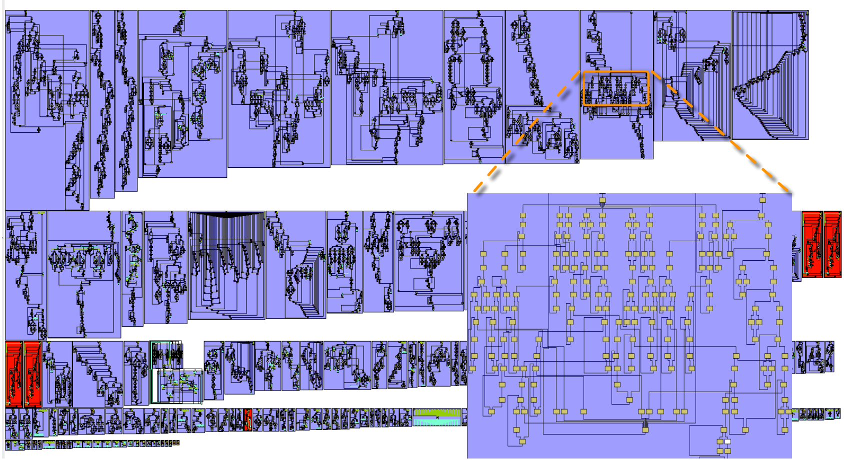


- Soundness = Over-Approximation
- Completeness = Under-Approximation
- Most detectors for C: neither sound nor complete

Existing Static Memory Leak Detectors

- Reason about flow of values on CFGs:
  - CLOUSEAU [PLDI' 03]
  - SATURN [FSE' 05]
  - CONTRADICTION [SAS' 06]
  - SPARROW [ISMM' 08]
  - CLANG

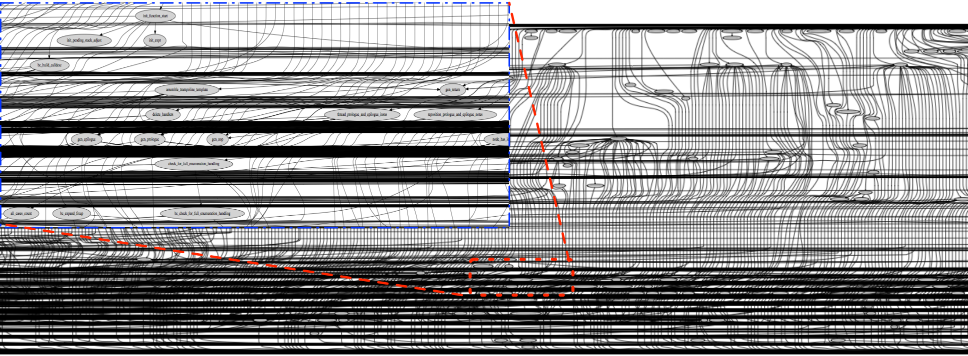
Whole-Program CFG of 300.twolf (20.5KLOC)



#functions: 194 #pointers: 20773 #loads/stores: 8657

Costly to reason about flow of values on CFGs!

Call Graph of 176.gcc (230.4KLOC)



#functions: 2256 #pointers: 134380 #loads/stores: 51543

Costly to reason about flow of values on CFGs!

Existing Static Memory Leak Detectors

- Reason about flow of values on CFGs:
  - CLOUSEAU [PLDI' 03]
  - SATURN [FSE' 05]
  - CONTRADICTION [SAS' 06]
  - SPARROW [ISMM' 08]
  - CLANG
- Reason about flow of values across def-use chains:
  - FASTCHECK [PLDI' 07] (for top-level pointers only)

SABER: Our Static Memory Leak Detector

Detecting Leaks Using Full-Sparse Value-Flow Analysis

- Leveraging recent advances on sparse pointer analysis
  - Semi-sparse (Hardekopf and Lin, POPL'09)
  - Level by Level (Yu, Xue, et al, CGO'10)
  - SPAS: (Sui, Ye and Xue, APLAS'11)
  - SFS (Hardekopf and Lin, CGO'11)
- Fully implemented in Open64
- Detects memory leaks in C programs

Sparse pointer analysis + sparse value-flow $\Longrightarrow$ Leak Detection

SABER vs. Other Static Leak Detectors

Leak Detector	Speed LOC/sec	Memory Leaks	False Positive Rate (%)
CONTRADICTION	300	26	56
CLANG	400	27	25
SPARROW	720	81	16
FASTCHECK	37,900	59	14
SABER	**10,220**	**83**	**19**

- Applied to the 15 SPEC2000 C programs
- The data for CONTRADICTION and SPARROW: their papers
- FASTCHECK and CLANG are open-source tools

SABER: Full-Sparse Value-Flow Analysis

- Value-Flow: tracking flow of values across def-use chains
  - Bootstrapped by recent advances on sparse pointer analysis
  - Maintaining significant less information
- Full-Sparse: both top-level and address-taken pointers
- On-Demandedness:
  - Analyzing only relevant portions of a program
  - Reasoning about the path conditions guarding the flow of a value only on the relevant portions — doing so on the entire CFG is costly!

Like most leak detectors, SABER is neither sound nor complete.

Outline

1. A motivating example
2. The SABER memory leak detector
3. Experimental results
4. Conclusion

A Motivating Example

```
void foo(){
  int *a; int **p;
  int **q = &a
  p = malloc(); // o
  *q = p;
  if(c){
    free(*q);
  }
  else{
   bar(p);
  }
}
void bar(int *r){
    free(r);
}
```

A Motivating Example

Phase 1: Pre-Analysis

```
void foo(){
  int *a; int **p;
  int **q = &a
  p = malloc(); // o
  *q = p;
  if(c){
    free(*q);
  }
  else{
    bar(p);
  }
}
void bar(int *r){
    free(r);
}
```

Points-to Info

$p \rightarrow o$

$q \rightarrow a$

$a \rightarrow o$

A Motivating Example

Phase 2: Full Sparse SSA

```
void foo(){
  int *a; int **p;
  int **q = &a
  p = malloc(); // o
  *q = p;       a=χ(a)
  if(c){
    free(*q);
  }
  else{
   bar(p);
  }
}
void bar(int *r){
     free(r);
}
```

Points-to Info

$p \rightarrow o$

$q \rightarrow a$

$a \rightarrow o$

A Motivating Example

Phase 2: Full Sparse SSA

```
void foo(){
  int *a; int **p;
  int **q = &a
  p = malloc(); // o
  *q = p;       a=χ(a)
  if(c){
    free(*q);    μ(a)
  }
  else{
    bar(p);
  }
}
void bar(int *r){
    free(r);
}
```

Points-to Info
p → o
q → a
a → o

A Motivating Example

Phase 2: Full Sparse SSA

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
    free(r1);
}
```

Points-to Info

$p \rightarrow o$

$q \rightarrow a$

$a \rightarrow o$

@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group

A Motivating Example

Phase 3: SVFG

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
    free(r1);
}
```

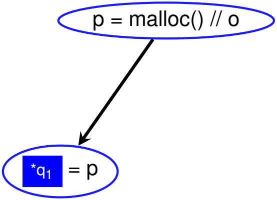
p = malloc() // o

@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group

A Motivating Example

Phase 3: SVFG

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
   bar(p1);
  }
}
 void bar(int *r1){
      free(r1);
 }
```

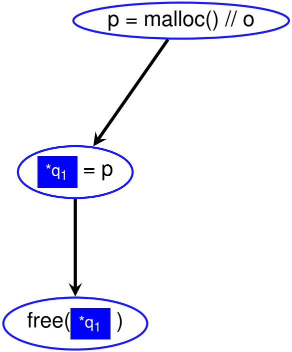


A Motivating Example

Phase 3: SVFG

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
    free(r1);
}
```

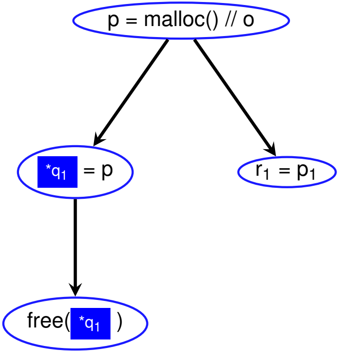
⟸



A Motivating Example

Phase 3: SVFG

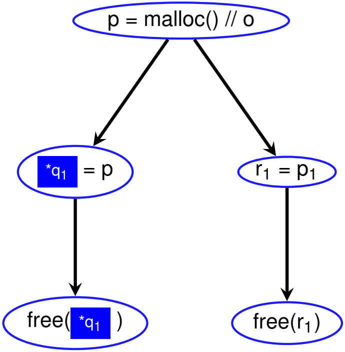
```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
    bar(p1);
  }
}
void bar(int *r1){
    free(r1);
}
```



A Motivating Example

Phase 3: SVFG

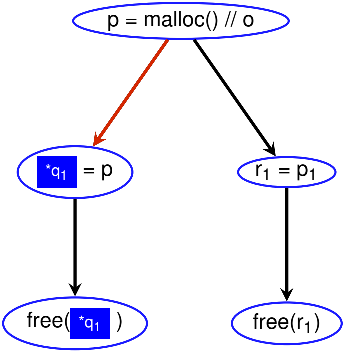
```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
    free(r1);
}
```



A Motivating Example

Phase 4: Reachability — (a) Forward

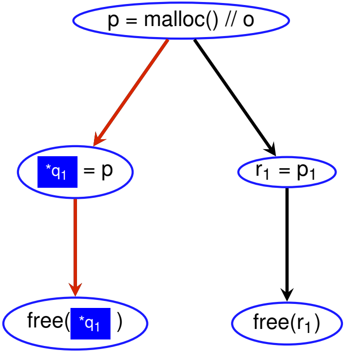
```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
     free(r1);
}
```



A Motivating Example

Phase 4: Reachability — (a) Forward

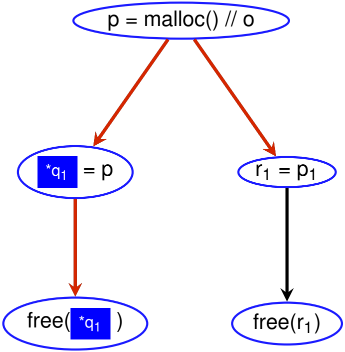
```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
    free(r1);
 }
```



A Motivating Example

Phase 4: Reachability — (a) Forward

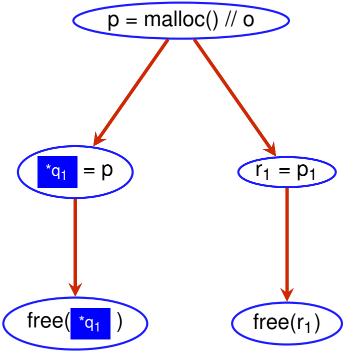
```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
    free(r1);
}
```



A Motivating Example

Phase 4: Reachability — (a) Forward

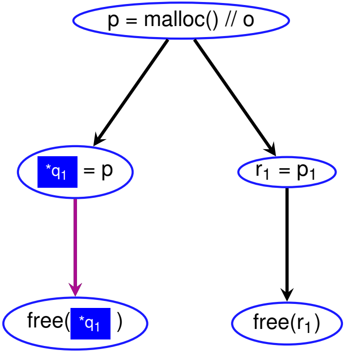
void foo(){
  int $*a_1$; int $**p$;
  int $**q_1$ = &a
  p_1 = malloc(); // o
  $\boxed{*q_1}$ = p_1; $a_2 = \chi(a_1)$
  if(c){
    free($\boxed{*q_1}$); $\mu(a_2)$
  }
  else{
   bar(p_1);
  }
}
void *bar*(int $*r_1$){
    free(r_1);
}



A Motivating Example

Phase 4: Reachability — (b) Backward

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;          a2=χ(a1)
  if(c){
    free(*q1);       μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
    free(r1);
}
```

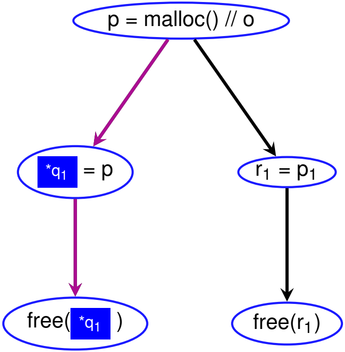


@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group

A Motivating Example

Phase 4: Reachability — (b) Backward

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
    free(r1);
 }
```



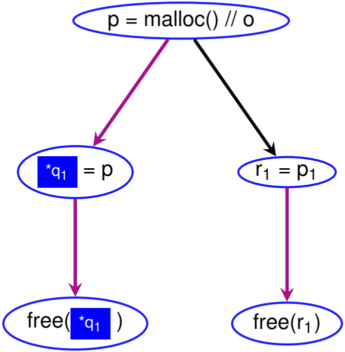
@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group

UNSW THE UNIVERSITY OF NEW SOUTH WALES

A Motivating Example

Phase 4: Reachability — (b) Backward

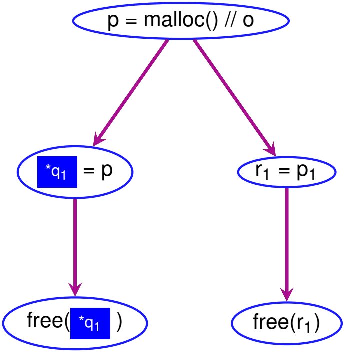
```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;          a2=χ(a1)
  if(c){
    free(*q1);       μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
     free(r1);
 }
```



A Motivating Example

Phase 4: Reachability — (b) Backward

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
    free(r1);
}
```

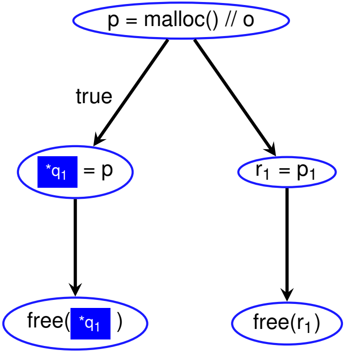


@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group

A Motivating Example

Phase 4: Reachability — (c) Guards

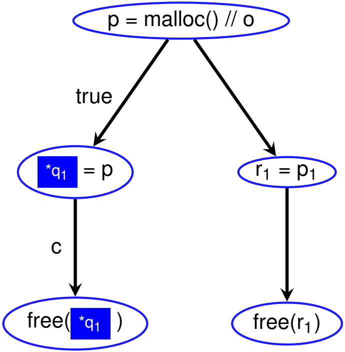
```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
    free(r1);
}
```



A Motivating Example

Phase 4: Reachability — (c) Guards

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;          a2=χ(a1)
  if(c){
    free(*q1);       μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
    free(r1);
}
```

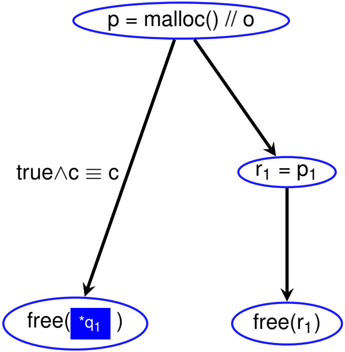


@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group

A Motivating Example

Phase 4: Reachability — (c) Guards

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;       a2=χ(a1)
  if(c){
    free( *q1 );    μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
    free(r1);
}
```

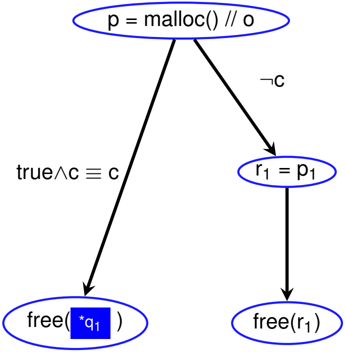


@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group

A Motivating Example

Phase 4: Reachability — (c) Guards

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;          a2=χ(a1)
  if(c){
    free( *q1 );     μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
    free(r1);
}
```

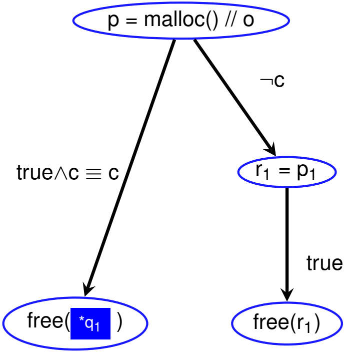


@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group

A Motivating Example

Phase 4: Reachability — (c) Guards

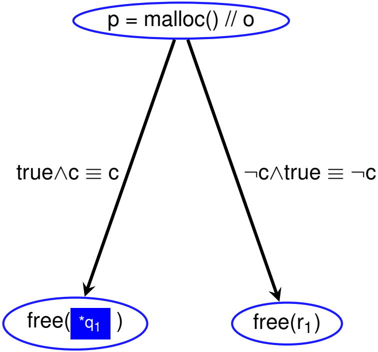
```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;          a2=χ(a1)
  if(c){
    free(*q1);       μ(a2)
  }
  else{
    bar(p1);
  }
}
void bar(int *r1){
    free(r1);
}
```



A Motivating Example

Phase 4: Reachability — (c) Guards

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free( *q1 );   μ(a2)
  }
  else{
   bar(p1);
  }
}
 void bar(int *r1){
     free(r1);
 }
```

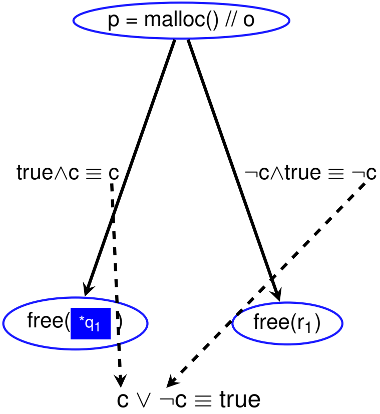


@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group

A Motivating Example

Phase 4: Reachability — (c) Guards

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;          a2=χ(a1)
  if(c){
    free(*q1);       μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
    free(r1);
}
```



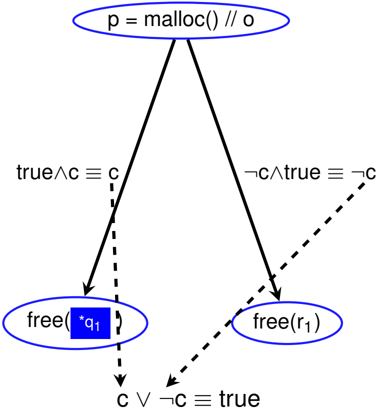
Safe! *o* reaches a free on all paths

@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group

A Motivating Example

Phase 4: Reachability — (c) Guards

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
  /* free(r1); */
}
```

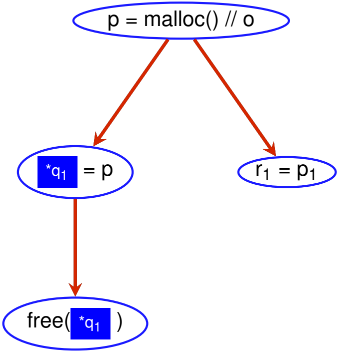


Safe! *o* reaches a free on all paths

A Motivating Example

Phase 4: Reachability — (a) Forward

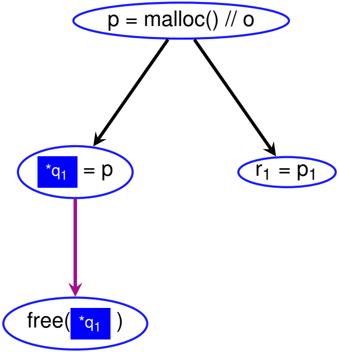
```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
  /* free(r1); */
}
```



A Motivating Example

Phase 4: Reachability — (b) Backward

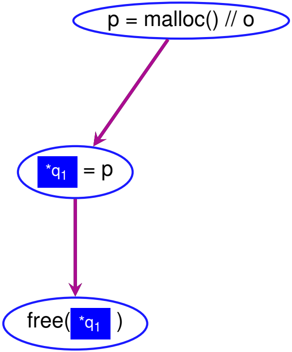
```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
    bar(p1);
  }
}
void bar(int *r1){
  /* free(r1); */
}
```



A Motivating Example

Phase 4: Reachability — (b) Backward

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
  /* free(r1); */
 }
```

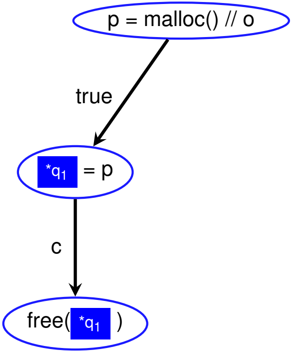


@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group

A Motivating Example

Phase 4: Reachability — (c) Guards

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
    bar(p1);
  }
}
void bar(int *r1){
  /* free(r1); */
}
```

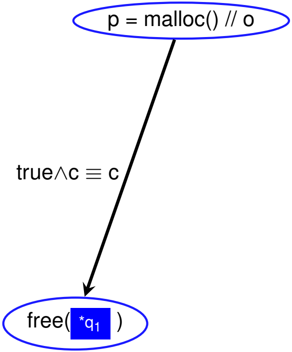


@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group

A Motivating Example

Phase 4: Reachability — (c) Guards

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free( *q1 );     μ(a2)
  }
  else{
   bar(p1);
  }
}
void bar(int *r1){
  /* free(r1); */
}
```

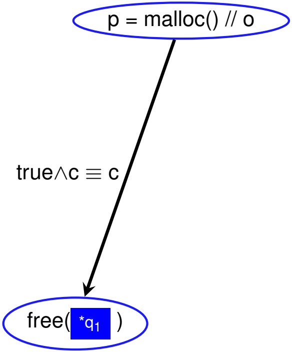


@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group

A Motivating Example

Phase 4: Reachability — (c) Guards

```
void foo(){
  int *a1; int **p;
  int **q1 = &a
  p1 = malloc(); // o
  *q1 = p1;        a2=χ(a1)
  if(c){
    free(*q1);     μ(a2)
  }
  else{
   bar(p1);
  }
}
 void bar(int *r1){
   /* free(r1); */
 }
```



$c \neq true$

Leak! *o* doesn't reach a free along path ¬c

Outline

1. A motivating example
2. The SABER memory leak detector
3. Experimental results
4. Conclusion

The SABER Framework

Phase 1: Perform Pre-Analysis	→	Phase 2: Build Full-Sparse SSA	→	Phase 3: Build SVFG	→	Phase 4: Detect Leaks via Graph Reachability on SVFG

- Full-sparse SSA = better memory SSA
- SVFG = Sparse Value-FLow Graph

Phase 1: Pre-Analysis

- Andersen's flow- and context-insensitive pointer analysis
  - Fields: offset-sensitive
  - Arrays: elements not distinguished
- Heap abstraction: malloc + malloc wrappers

Phase 2: Build Full-Sparse SSA

- Partition memory into regions
- Add μ's (MAY-USEs) and χ's (MAY-DEFs)
- Build SSA by distinguishing local and nonlocal variables — any value flowing into globals not analysed (assumed safe)

2: Building Memory SSA (loads and stores)

```
void foo(){
  ...

 r = *p

  ...
 *q = s




}
```

2: Building Memory SSA (loads and stores)

```
void foo(){
  ...

 r = *p

  ...
 *q = s

}
```

POINTS-TO::

$p \rightarrow \{ \quad x, y \quad ,v \quad \}$

$q \rightarrow \{ \quad x, z \quad ,w \quad \}$

2: Building Memory SSA (loads and stores)

```
void foo(){
  ...

  r = *p

  ...
  *q = s



}
```

POINTS-TO::

$p \rightarrow \{\ x, y\ ,v\ \}$

$q \rightarrow \{\ x, z\ ,w\ \}$

x, y, z: nonlocal in foo
v, w: locally declared in foo

2: Building Memory SSA (loads and stores)

```
void foo(){
  ...

  r = *p

  ...
  *q = s



}
```

REGION PARTITIONING:

POINTS-TO::

$p \rightarrow \{\, R(x, y), v \,\}$

$q \rightarrow \{\, R(x, z), w \,\}$

x, y, z: nonlocal in foo
v, w: locally declared in foo

2: Building Memory SSA (loads and stores)

void foo(){

$\ldots$

$\mu(v)\ \mu(\boxed{R(x, y)})$

r = *p

$\ldots$

*q = s

$w = \chi(w)$

$\boxed{R(x, z)} = \chi(\boxed{R(x, z)})$

}

REGION PARTITIONING:

POINTS-TO::

p $\rightarrow$ { $\boxed{R(x, y)}$,v }

q $\rightarrow$ { $\boxed{R(x, z)}$,w }

x, y, z: nonlocal in foo
v, w: locally declared in foo

2: Building Memory SSA (loads and stores)

void foo(){

…

$\mu(v)\ \mu(\boxed{R(x,y)})$

r = *p

…

*q = s

$w = \chi(w)$

$\boxed{R(x,z)} = \chi(\boxed{R(x,z)})$

}

$\boxed{R(x,y)} \cap \boxed{R(x,z)} \neq \emptyset$

REGION PARTITIONING:

POINTS-TO::

$p \rightarrow \{ \boxed{R(x,y)}, v \}$

$q \rightarrow \{ \boxed{R(x,z)}, w \}$

x, y, z: nonlocal in foo
v, w: locally declared in foo

2: Building Memory SSA (loads and stores)

void foo(){

$\ldots$

$\mu(v)\ \mu(\boxed{R(x,y)})\ \ \mu(\boxed{R(x,z)})$

r = *p

$\ldots$

*q = s

$w = \chi(w)$

$\boxed{R(x,z)} = \chi(\boxed{R(x,z)})$

$\boxed{R(x,y)} = \chi(\boxed{R(x,y)})$

}

$\boxed{R(x,y)} \cap \boxed{R(x,z)} \neq \emptyset$

REGION PARTITIONING:

POINTS-TO::

$p \rightarrow \{\ \boxed{R(x,y)}, v\ \}$

$q \rightarrow \{\ \boxed{R(x,z)}, w\ \}$

x, y, z: nonlocal in foo
v, w: locally declared in foo

2: Building Memory SSA (callsites)

$$
\begin{array}{l}
\text{void foo()}\{ \\
\quad \ldots \\
\quad \mu(v)\ \mu(\ R(\text{x, y})\ )\ \ \mu(\ R(\text{x, z})\ ) \\
\quad \text{r = *p} \\
\quad \ldots \\
\quad \text{*q = s} \\
\quad w = \chi(w) \\
\quad\ R(\text{x, z}) = \chi(\ R(\text{x, z})\ ) \\
\quad\ R(\text{x, y}) = \chi(\ R(\text{x, y})\ ) \\
\\
\quad \text{bar}(\ldots) \\
\\
\}
\end{array}
$$

x, y, z: nonlocal in foo
v, w: locally declared in foo

2: Building Memory SSA (callsites)

```
void foo(){
  ...
  μ(v) μ( R(x, y) ) μ( R(x, z) )
  r = *p
  ...
  *q = s
  w = χ(w)
   R(x, z) = χ( R(x, z) )
   R(x, y) = χ( R(x, y) )


  bar(...)


}
```

CALLSITE REF/MOD:
REF: v
MOD: $R(z)$

x, y, z: nonlocal in foo
v, w: locally declared in foo

2: Building Memory SSA (callsites)

$$
\begin{array}{l}
\text{void foo()}\{ \\
\quad \ldots \\
\quad \mu(v)\ \mu(\ R(\text{x, y})\ )\ \ \mu(\ R(\text{x, z})\ ) \\
\quad \text{r = *p} \\
\quad \ldots \\
\quad \text{*q = s} \\
\quad w = \chi(w) \\
\quad R(\text{x, z}) = \chi(\ R(\text{x, z})\ ) \\
\quad R(\text{x, y}) = \chi(\ R(\text{x, y})\ ) \\
\\
\quad \mu(v) \\
\quad \text{bar(}\ldots\text{)} \\
\\
\}
\end{array}
$$

CALLSITE REF/MOD:

REF: v

MOD: $R(\text{z})$

x, y, z: nonlocal in foo

v, w: locally declared in foo

2: Building Memory SSA (callsites)

```
void foo(){
 ...
 μ(v) μ( R(x, y) )  μ( R(x, z) )
 r = *p
 ...
 *q = s
 w = χ(w)
  R(x, z) = χ( R(x, z) )
  R(x, y) = χ( R(x, y) )


 μ(v)
 bar(...)
  R(z) = χ( R(z) )
}
```

CALLSITE REF/MOD:
REF: v
MOD: $R(z)$

x, y, z: nonlocal in foo
v, w: locally declared in foo

2: Building Memory SSA (callsites)

```
void foo(){
  ...
  μ(v) μ( R(x, y) )  μ( R(x, z) )
  r = *p
  ...
  *q = s
  w = χ(w)
   R(x, z) = χ( R(x, z) )
   R(x, y) = χ( R(x, y) )


  μ(v)
  bar(...)
   R(z) = χ( R(z) )
}
```

$\boxed{R(x,z)} \cap \boxed{R(z)} \neq \emptyset$

CALLSITE REF/MOD:
REF: v
MOD: $\boxed{R(z)}$

x, y, z: nonlocal in foo
v, w: locally declared in foo

2: Building Memory SSA (callsites)

```
void foo(){
  ...
  μ(v) μ( R(x, y) )  μ( R(x, z) )  μ( R(z) )
  r = *p
  ...
  *q = s
  w = χ(w)
  R(x, z) = χ( R(x, z) )
  R(x, y) = χ( R(x, y) )
  R(z) = χ( R(z) )

  μ(v)
  bar(...)
  R(z) = χ( R(z) )
}
```

$R(x, z) \cap R(z) \neq \emptyset$

CALLSITE REF/MOD:
REF: v
MOD: $R(z)$

x, y, z: nonlocal in foo
v, w: locally declared in foo

2: Building Memory SSA (callsites)

```
void foo(){
  ...
  μ(v) μ( R(x, y) )  μ( R(x, z) )  μ( R(z) )
  r = *p
  ...
  *q = s
  w = χ(w)
  R(x, z) = χ( R(x, z) )
  R(x, y) = χ( R(x, y) )
  R(z) = χ( R(z) )

  μ(v)
  bar(...)
  R(z) = χ( R(z) )
}
```

$R(x, z) \cap R(z) \neq \emptyset$

CALLSITE REF/MOD:
REF: v
MOD: $R(z)$

x, y, z: nonlocal in foo
v, w: locally declared in foo

Finally, build SSA using a standard algorithm

@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group

Phase 3: Build SVFG

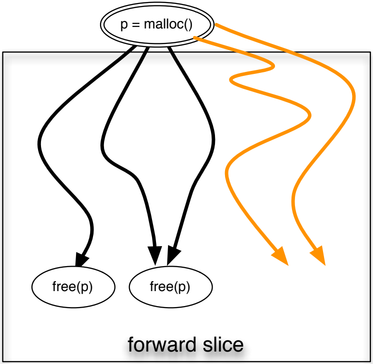
Rule	Statement (SSA)	Value-Flow Edges
ASN	$p_i = q_j$	$\widehat{p_i} \leftarrow \widehat{q_j}$
MU	$\mu(v_t)$ $p_i = *q_j$	$\widehat{p_i} \leftarrow \widehat{v_t}$
CHI	$*p_i = q_j$ $v_s = \chi(v_t)$	$\widehat{v_s} \leftarrow \widehat{q_j}$, $\widehat{v_s} \leftarrow \widehat{v_t}$
PHI	$p_i=\phi(q_j,q_k)$	$\widehat{p_i} \leftarrow \widehat{q_j}$, $\widehat{p_i} \leftarrow \widehat{q_k}$
CALL (at a callsite c for a callee g)	$\mu(v_m)$ r_i=g$(\ldots, a_k, \ldots)$ $v_s = \chi(v_t)$	$(1) U_c^g(v_m) \leftarrow \widehat{\mu(v_m)}$, $(2)\ \widehat{\mu(v_m)} \xleftarrow{(_c^g} \widehat{v_m}$ $(3) FP(a_k) \xleftarrow{(_c^g} \widehat{a_k}$, $(4)\ \widehat{r_i} \xleftarrow{)_c^g} RET(r_i)$ $(5)\ \widehat{v_s} \xleftarrow{)_c^g} D_c^g(v_s)$, $(6)\ \widehat{v_s} \leftarrow \widehat{v_t}$

- A node: a def site (and μ's)
- An edge from $\widehat{p}$ to $\widehat{q}$: a value flows from $\widehat{p}$ to $\widehat{q}$

Phase 4: Detect Leaks via Graph Reachability

(a): Compute a forward slice from an allocation site

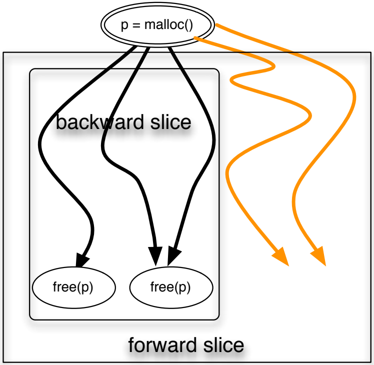
- Performing graph reachability on SVFG
- Matching call and return (context-sensitively).



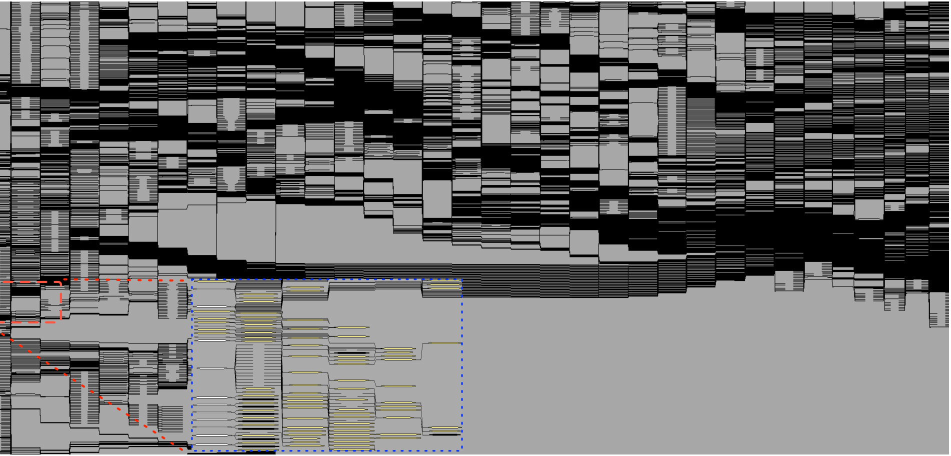
Phase 4: Detect Leaks via Graph Reachability

(b) Refining a forward slice into a backward slice

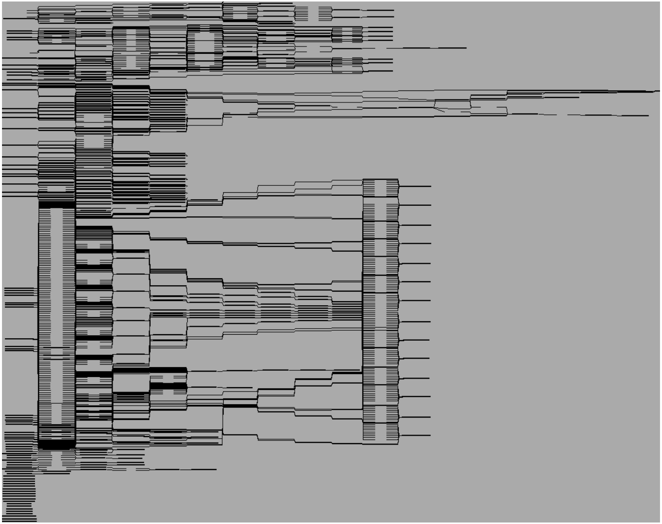
- Including only the SVFG nodes reaching a sink (or free site)
- Computing guards (or path conditions) on-demand



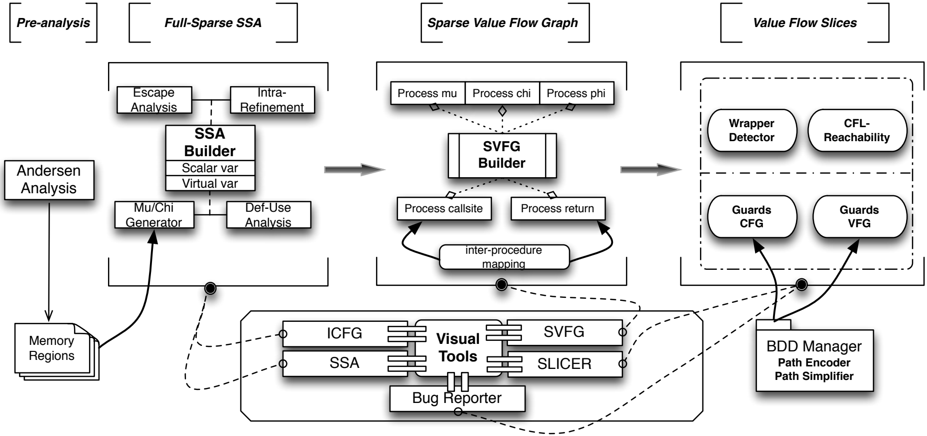
Forward Slices in 175.vpr



The Refined Backward Slices in 175.vpr



The SABER Implementation in Open64



20 KLOC with the core part coded in 5K LOC

Outline

1. A motivating example
2. The SABER memory leak detector
3. Experimental results
4. Conclusion

Bug Counts and Analysis Times

Program	Size (KLOC)	Time (secs)	#Bugs	#False Alarms
ammp	13.4	0.55	20	0
art	1.2	0.01	1	0
bzip2	4.7	0.04	1	0
crafty	21.2	0.83	0	0
equake	1.5	0.04	0	0
gap	71.5	4.00	0	0
gcc	230.4	20.88	40	5
gzip	8.6	0.08	1	0
mcf	2.5	0.03	0	0
mesa	61.3	10.10	7	4
parser	11.4	0.28	0	0
perlbmk	87.1	18.52	8	4
twolf	20.5	2.12	5	0
vortex	67.3	2.90	0	4
vpr	17.8	0.31	0	3
bash	100.0	22.03	8	2
httpd	128.1	10.65	0	0
icecast	22.3	5.54	12	5
sendmail	115.2	32.97	2	0
wine	1338.1	390.7	106	21
Total	2324.1	522.58	211	48

Conditional Leaks in mesa

```
344:  static struct gl_texture_image *
        image_to_texture( GLcontext *ctx,
        const struct gl_image *image){

349:       struct gl_texture_image *texImage;
362:       texImage = gl_alloc_texture_image();
            ...
385:       texImage->Data = (GLubyte *)malloc
                        ( numPixels * components );
            ...
451:        switch (texImage->Format) {
452:              case GL_ALPHA:
                  ...
454:              break;
455:              case GL_LUMINANCE:
                  ...
457:              break;
476:              default:
478:                    return NULL;
             }
            ...
786:        return texImage;
        }
```

@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group
THE UNIVERSITY OF NEW SOUTH WALES

Conditional Leaks in wine

```
      //ungif.c
890:  GifFileType *
891:  DGifOpen(void *userData,
          InputFunc readFunc) {
898:  GifFile = malloc(sizeof(GifFileType));
      ...
905:  Private = malloc(sizeof(GifFilePrivateType));
911:  GifFile->Private = (void*)Private;
      ...
938:  return GifFile;
      }

944:  int
945:  DGifCloseFile(GifFileType * GifFile) {
947:      GifFilePrivateType *Private;
          ...
952:      Private =  GifFile->Private;
964:      free(Private);
972:      free(GifFile);
974:      return GIF_OK;
      }
```

```
        //olepicture.c
1021:   static HRESULT OLEPictureImpl_LoadGif
            (OLEPictureImpl *This, BYTE *xbuf)
         {
1006:    GifFileType    *gif;
             ...
1021:    gif = DGifOpen((void*)&gd, _gif_inputfunc);
         ...
1030:    if (gif->ImageCount<1){
1031:     FIXME("GIF stream does
                not have images inside?\n");
1032:         return E_FAIL;
         }
                ...
1194:    DGifCloseFile(gif);
1195:    HeapFree(GetProcessHeap(),0,bytes);
1196:    return S_OK;
         }
```

Conditional Leaks in icecast

```
     //avl.c
42:  avl_node *avl_node_new (void *key,avl_node *parent)
     {
45:      avl_node * node =  malloc (sizeof (avl_node));
47:      if (!node) {
48:          return NULL;
49:      }else {
50:          node->parent = parent;
51:          node->key = key;
58:          return node;
         }
     }

116:  int avl_insert (avl_tree * ob, void * key){
120:        avl_node* node = avl_node_new(key, ob->root);
121:        if (!node) {
122:          return -1;
123:        } else {
              ...
127:        }
128:  }
```

```
      //auth_htpasswd.c
120:  static void htpasswd_recheckfile
            (htpasswd_auth_state *htpasswd){
123:    avl_tree *new_users;
157:    new_users = avl_tree_new (compare_users, NULL);
        ...
159:    while (get_line(passwdfile, line, MAX_LINE_LEN))
        {
161:      int len;
162:      htpasswd_user *entry;
          ...
174:      entry = calloc (1, sizeof (htpasswd_user));
176:      entry->name = malloc (len);
          ...
180:      avl_insert (new_users, entry);
        }
      }
```

Limitations

False Positives (FPs)

- Path correlations
- Approximating loops
- Reference accounting
- Multi-dimensional arrays
- External unknown Calls
- Over-approximating pointer information

Path Correlations False Positive (from perl)

```
char smallbuf[256];
char *tmpbuf;
STRLEN tmplen;

if (tmplen < sizeof(smallbuf))
    tmpbuf = smallbuf;
else
    New(603, tmpbuf, char);

if (tmpbuf != smallbuf)
    Safefree(tmpbuf);
// report a false alarm for missing a free at the else branch
```

Handling Loops False Positive (from vortex)

```
int start = 1;
int *res;
while(i <= 10){
   if(start){
     res = malloc();
     start = 0;
   }
   else{
     if (rest != null)
       free(res);
   }
}
// report a false alarm
```

- SABER analyses only the first N iterations (= 1 in ISSTA'12)
- Still related to path correlations inside

@ Yulei Sui, ISSTA 2012, 19/7/2012
Programming Languages and Compilers Group

Reference Counting False Positive (from mesa)

```
void foo() {
    p = malloc();
    q = malloc();
    assignment(&p,q);
}
void assignment(int **x, int *y) {
   if ((*x)->refcount == 1)
     free(*x);
   // reports a false alarm at the else branch
   ...
}
```

Multi-dimensional Arrays (from vpr)

```
void** alloc_matrix(){
    char** cptr = (char**) malloc(10);
    for(int i = 0; i < 10; i++)
        cptr[i] = (char*) malloc(1);      //report a false alarm
    return ((void**)cptr);
}
void free_matrix(void ** matrix){
    for(int i = 0; i < 10; i++)
        free(matrix[i]);
    free(matrix);
}
int foo(){
   char** dir_list = (char**)alloc_matrix();
   free_matrix(dir_list);
}
```

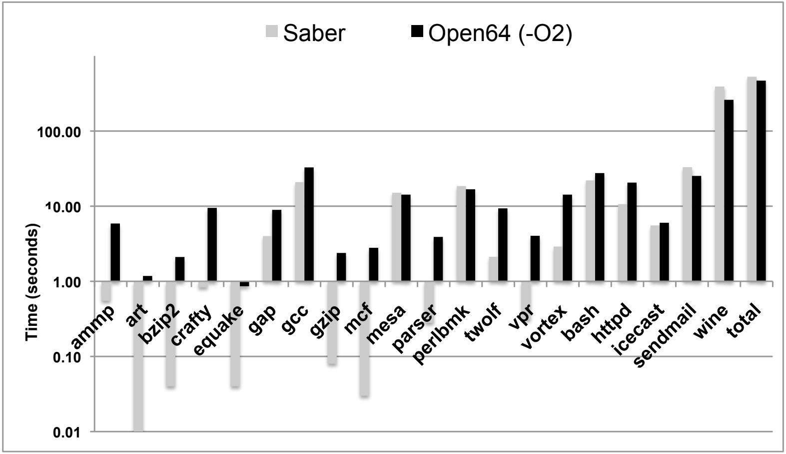
- Use `*cptr` to represent `cptr[i]` and `cptr[j]`
- Fooled to believe that cptr is overwritten inside the first loop

External Calls False Positive (from perl)

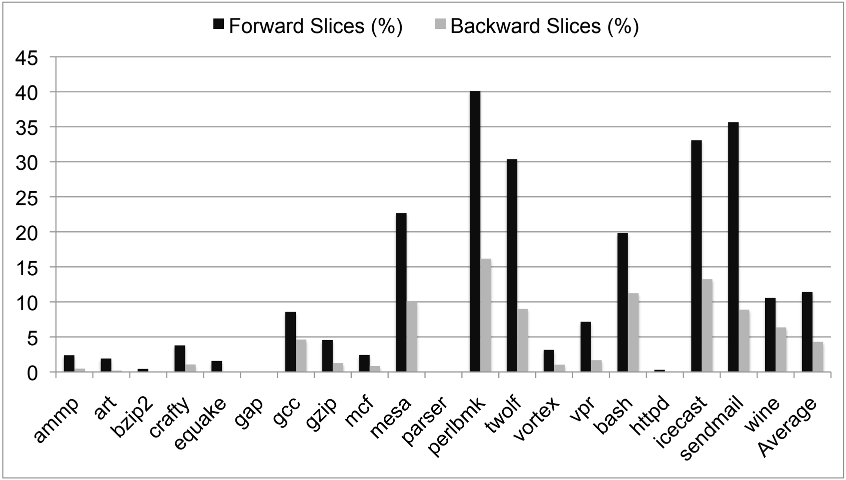
```
Newz(899,newargv,PL_origargc+3,char*); // allocate newargv
newargv[0] = ipath;
//external call to unknown code
PerlProc_execv(ipath, newargv); // report a false alarm
```

- Cannot analyze some non-existing code
- Report a false alarm to be safe

Comparable with Compilation Times



Traversed SVFG Nodes in Slices



Conclusion

- Value-Flow: tracking flow of values across def-use chains
  - Bootstrapped by recent advances on sparse analysis
  - Maintaining significant less information
- Full-Sparse: both top-level and address-taken pointers
- On-Demandedness:
  - Analyzing only relevant portions of a program
  - Reasoning about the path conditions guarding the flow of a value only on the relevant portions

Backup Slices

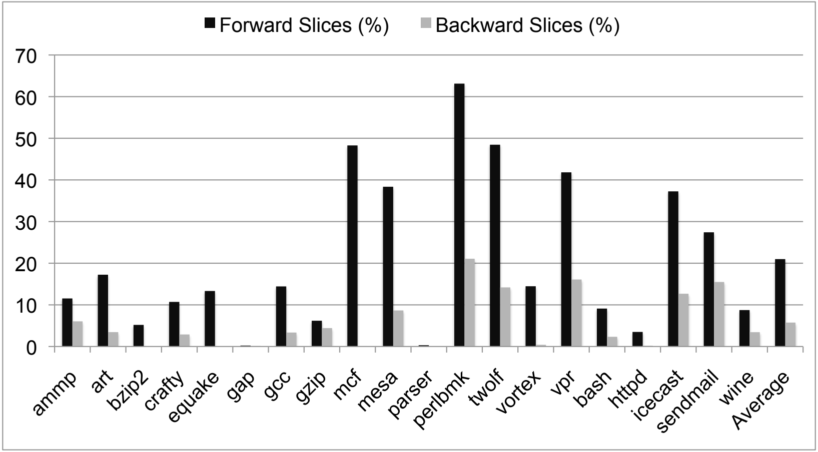
Benchmark Statistics

Program	#Functions	#Pointers	#Loads/Stores	#Alloc Sites	#Free Sites
ammp	182	9829	1636	37	30
art	29	600	103	11	1
bzip2	77	1672	434	10	4
crafty	112	11883	3307	12	16
equake	30	1203	408	29	0
gap	857	61435	16841	2	1
gcc	2256	134380	51543	161	19
gzip	113	3004	586	3	3
mcf	29	1317	526	4	3
mesa	1109	44582	17302	82	76
parser	327	8228	2597	1	0
perlbmk	1079	54816	16885	148	2
twolf	194	20773	8657	185	1
vortex	926	40260	11256	9	3
vpr	275	7930	2160	130	68
bash	2700	17830	6855	112	58
httpd	3000	60027	18450	21	18
icecast	603	15098	9779	235	235
sendmail	2656	107242	22191	296	136
wine	77829	1330840	137409	515	231

Leak Detection Statistics

Program	#Node in SVFG	#Functions Included (%)		#SVFG's Nodes Included (%)	
		$\mathcal{F}_{src}$'s	$\mathcal{B}_{src}$'s	$\mathcal{F}_{src}$'s	$\mathcal{B}_{src}$'s
ammp	72362	11.54%	6.04%	2.38%	0.48%
art	2061	17.24%	3.45%	1.92%	0.20%
bzip2	4943	5.19%	0.08%	0.43%	0.01%
crafty	56750	10.71%	2.89%	3.79%	1.07%
equake	3071	13.33%	0.00%	1.57%	0.00%
gap	277614	0.23%	0.12%	0.01%	0.00%
gcc	838373	14.43%	3.35%	8.59%	4.63%
gzip	6048	6.19%	4.42%	4.55%	1.25%
mcf	8160	48.28%	0.06%	2.42%	0.825%
mesa	1427669	38.36%	8.66%	22.67%	10.10%
parser	29016	0.31%	0.00%	0.01%	0.00%
perlbmk	698646	63.12%	21.07%	40.12%	16.18%
twolf	193074	48.45%	14.19%	30.38%	9.01%
vortex	146047	14.47%	0.41%	3.16%	1.05%
vpr	24814	41.82%	16.09%	7.18%	1.67%
bash	32129	9.11%	2.33%	19.87%	11.23%
httpd	176528	3.50%	0.17%	0.31%	0.05%
icecast	41474	37.25%	12.67%	33.07%	13.23%
sendmail	824181	27.42%	15.49%	35.67%	8.90%
wine	2928148	8.75%	3.44%	10.59%	6.36%

Traversed Functions in Forward/Backward Slices



Backup Slice: Computing Guards On-Demand for Loops

void `foo`() {

...

B_0 v_0 = ...;

B_1 for(k=0; k$<$10; k++){

B_2 if(n $<$ 2)

B_3 ...

else

B_4 return;

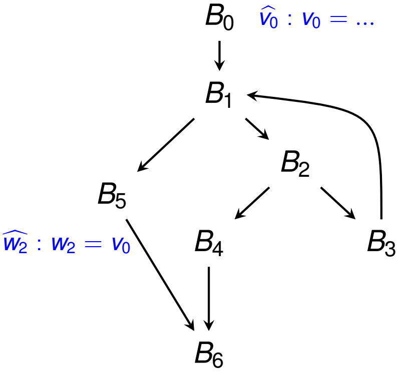
}

B_5 w_2 = v_0; free(w_2)

B_6 }

$C_1 \xleftarrow{encode} (\text{k} < 10)$

$C_2 \xleftarrow{encode} (\text{n} < 2)$



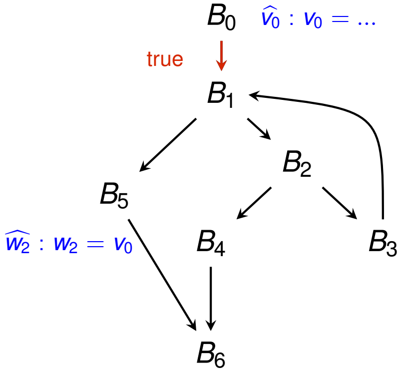
$cfguard(\widehat{v_0}, \widehat{w_2})$ =

Backup Slice: Computing Guards On-Demand for Loops

void `foo`() {
...
B_0 v_0 = ...;
B_1 for(k=0; k<10; k++){
B_2 if(n < 2)
B_3 ...
else
B_4 return;
}
B_5 w_2 = v_0; free(w_2)
B_6 }

$$C_1 \xleftarrow{encode} (k < 10)$$

$$C_2 \xleftarrow{encode} (n < 2)$$



$cfguard(\widehat{v_0}, \widehat{w_2})$ =

Backup Slice: Computing Guards On-Demand for Loops

void `foo`() {

...

B_0 v_0 = ...;

B_1 for(k=0; k<10; k++){

B_2 if(n $<$ 2)

B_3 ...

else

B_4 return;

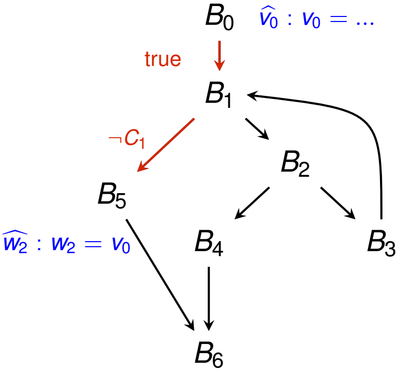
}

B_5 w_2 = v_0; free(w_2)

B_6 }

$C_1 \xleftarrow{encode} (k < 10)$

$C_2 \xleftarrow{encode} (n < 2)$



$cfguard(\widehat{v_0}, \widehat{w_2})$ =

Backup Slice: Computing Guards On-Demand for Loops

void `foo`() {

...

B_0 v_0 = ...;

B_1 for(k=0; k$<$10; k++){

B_2 if(n $<$ 2)

B_3 ...

else

B_4 return;

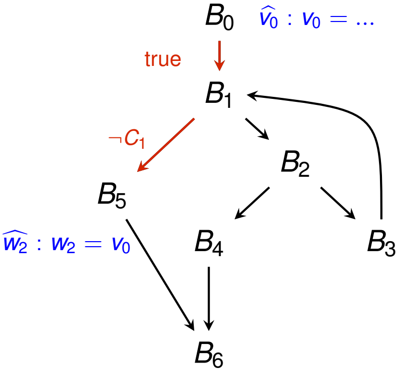
}

B_5 $\mathrm{w}_2 = \mathrm{v}_0$; free($\mathrm{w}_2$)

B_6 }

$C_1 \xleftarrow{encode} (\mathrm{k} < 10)$

$C_2 \xleftarrow{encode} (\mathrm{n} < 2)$



$cfguard(\widehat{v_0}, \widehat{w_2}) = \neg C_1$

Backup Slice: Computing Guards On-Demand for Loops

void `foo`() {

...

B_0 v_0 = ...;

B_1 for(k=0; k$<$10; k++){

B_2 if(n $<$ 2)

B_3 ...

else

B_4 return;

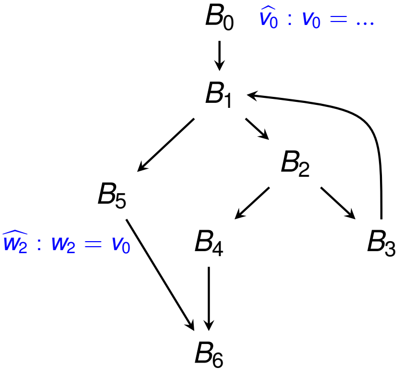
}

B_5 w_2 = v_0; free(w_2)

B_6 }

$$C_1 \xleftarrow{encode} (k < 10)$$

$$C_2 \xleftarrow{encode} (n < 2)$$



$$cfguard(\widehat{v_0}, \widehat{w_2}) = \neg C_1$$

Backup Slice: Computing Guards On-Demand for Loops

void `foo`() {

...

B_0 v_0 = ...;

B_1 for(k=0; k<10; k++){

B_2 if(n $<$ 2)

B_3 ...

else

B_4 return;

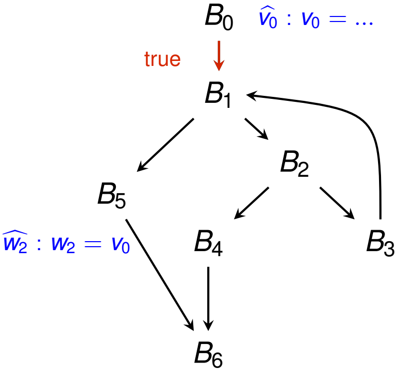
}

B_5 $w_2 = v_0$; free(w_2)

B_6 }

$$C_1 \xleftarrow{encode} (k < 10)$$

$$C_2 \xleftarrow{encode} (n < 2)$$



$$cfguard(\widehat{v_0}, \widehat{w_2}) = \neg C_1$$

Backup Slice: Computing Guards On-Demand for Loops

void foo() {

...

B_0 v_0 = ...;

B_1 for(k=0; k<10; k++){

B_2 if(n $<$ 2)

B_3 ...

else

B_4 return;

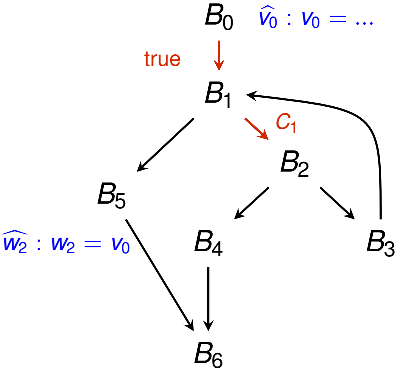
}

B_5 w_2 = v_0; free(w_2)

B_6 }

$C_1 \xleftarrow{encode} (k < 10)$

$C_2 \xleftarrow{encode} (n < 2)$



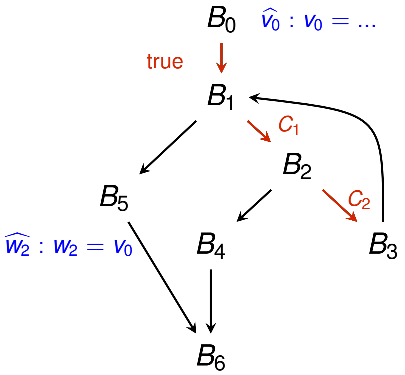
$cfguard(\widehat{v_0}, \widehat{w_2}) = \neg C_1$

Backup Slice: Computing Guards On-Demand for Loops

void `foo`() {
...
B_0 v_0 = ...;
B_1 for(k=0; k<10; k++){
B_2 if(n $<$ 2)
B_3 ...
else
B_4 return;
}
B_5 w$_2$ = v$_0$; free(w$_2$)
B_6 }

$$C_1 \xleftarrow{encode} (\mathrm{k} < 10)$$

$$C_2 \xleftarrow{encode} (\mathrm{n} < 2)$$



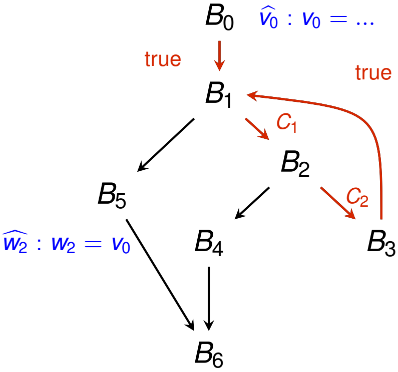
$$cfguard(\widehat{v_0}, \widehat{w_2}) = \neg C_1$$

Backup Slice: Computing Guards On-Demand for Loops

void `foo`() {
  ...
B_0 v_0 = ...;
B_1 for(k=0; k<10; k++){
B_2 if(n $<$ 2)
B_3 ...
    else
B_4 return;
  }
B_5 w_2 = v_0; free(w_2)
B_6 }

$C_1 \xleftarrow{encode} (k < 10)$

$C_2 \xleftarrow{encode} (n < 2)$



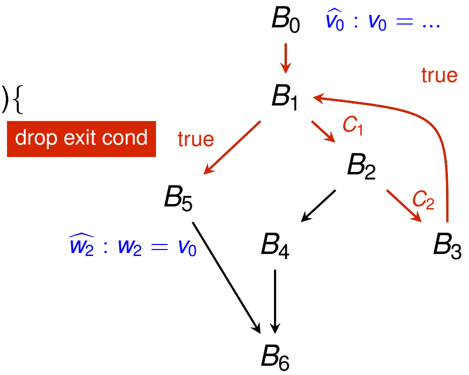
$cfguard(\widehat{v_0}, \widehat{w_2}) = \neg C_1$

Backup Slice: Computing Guards On-Demand for Loops

```
      void foo() {
         ...
B0  v0 = ...;
B1    for(k=0; k<10; k++){
B2        if(n < 2)
B3            ...
          else
B4            return;
        }
B5    w2 = v0; free(w2)
B6  }
```

$C_1 \xleftarrow{encode} (k < 10)$

$C_2 \xleftarrow{encode} (n < 2)$



$cfguard(\widehat{v_0}, \widehat{w_2}) = \neg C_1$

Backup Slice: Computing Guards On-Demand for Loops

void `foo`() {

...

B_0 $v_0 = \ldots;$

B_1 for(k=0; k<10; k++){

B_2 if(n $<$ 2)

B_3 ...

else

B_4 return;

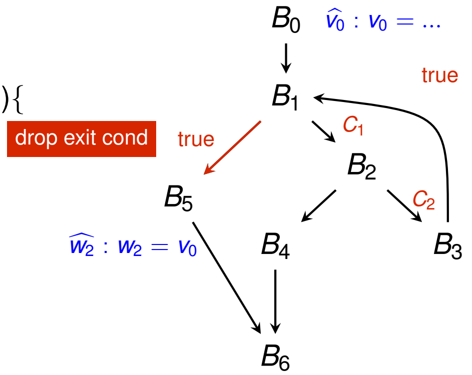
}

B_5 w$_2$ = v$_0$; free(w$_2$)

B_6 }

$C_1 \xleftarrow{encode} (\mathrm{k} < 10)$

$C_2 \xleftarrow{encode} (\mathrm{n} < 2)$



$$cfguard(\widehat{v_0}, \widehat{w_2}) = \neg C_1 \vee (C_1 \wedge C_2)$$