Accelerating Dynamic Detection of Uses of Undefined Values with Static Value-Flow Analysis

Ding Ye, Yulei Sui, Jingling Xue

Programming Languages and Compilers Group
School of Computer Science and Engineering
University of New South Wales, Australia

Never Stand Still
Faculty of Engineering
Computer Science and Engineering

February 18, 2014

Outline

- What are undefined values?
- Related work and our contributions
- Methodology
- Evaluation
- Conclusion

Use of undefined values

- Undefined values are caused by memory allocations without initialization
    - Stack variables and malloc() heaps in C
- Definedness is transitive
- May cause serious problems when used by some critical operations
    - Conditional jumps
    - Pointer dereferences

undefined / defined

```
void foo() {
     int a, b, c;
     a = 1 + 2;
     b = 0;
     ...
     b = c + 3;
l1:  if (a > b)
        ...
     ...
     int *p;
l2:  *p = a;
}
```

Outline

- What are undefined values?
- Related work and our contributions
- Methodology
- Evaluation
- Conclusion

Dynamic analysis

- Instrumentation via shadow memory

- Binary-based instrumentation
  - Insert code on the binary
  - e.g. Valgrind (>10X slowdown)

- Source-based instrumentation
  - Insert code at compile-time
  - e.g. MSan (typical 3X slowdown)

undefined / defined

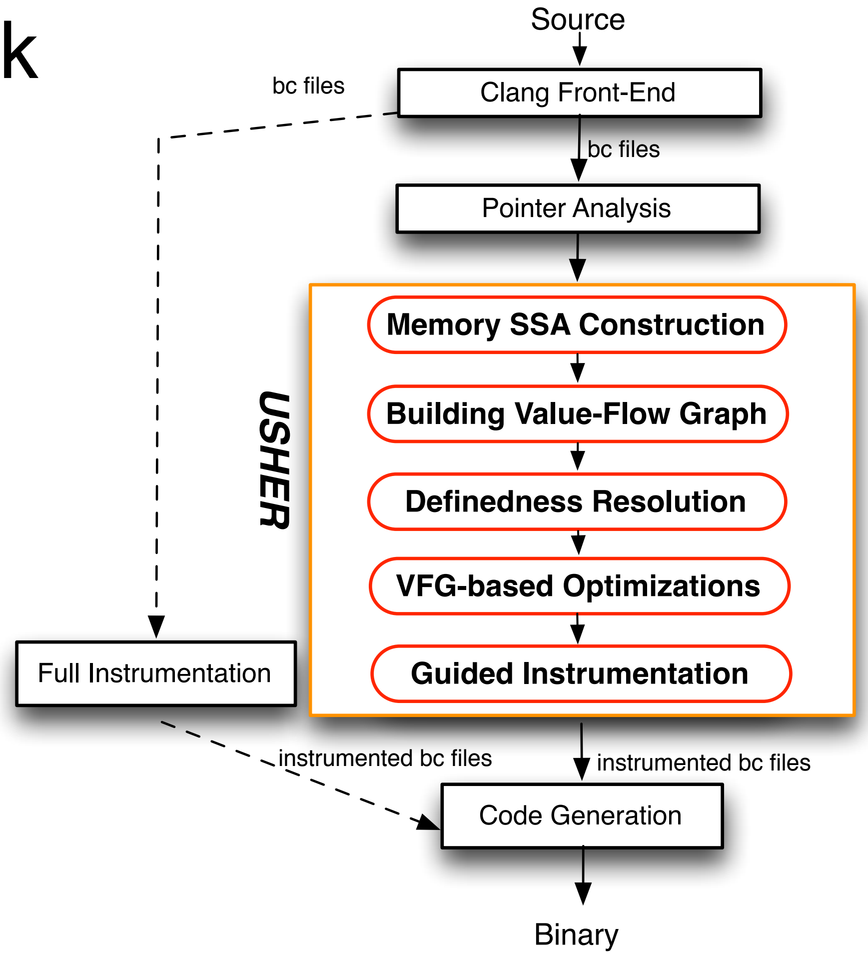
void foo() {	
int a, b, c;	**a_s= F; b_s= F; c_s= F;**
a = 1 + 2;	**a_s = T & T;**
b = 0;	**b_s= T;**
...	**...**
b = c + 3;	**b_s= c_s & T;**
I1: if (a > b)	**check (a_s& b_s);**
...	**...**
...	**...**
int *p;	**p_s = F;**
I2: *p = a;	**check (p_s);**
}	

Other analysis

- Static analysis
  - Dataflow analysis
  - Compilers (gcc, clang)
  - HDL, typestate verification, IFDS
- Static + dynamic
  - Nguyen *et al. CC '03*
    - For Fortran (5X slowdown)
  - Necula *et al. TOPLAS '05*
    - CCured applies only to pointers (requiring source code modification)

The *Usher* framework

- Instrumentation
  - Full *vs* Guided (selective)



Source

Clang Front-End

bc files

bc files

Pointer Analysis

USHER

Memory SSA Construction

Building Value-Flow Graph

Definedness Resolution

VFG-based Optimizations

Guided Instrumentation

Full Instrumentation

instrumented bc files

instrumented bc files

Code Generation

Binary

Our contributions

- Usher is a new static + dynamic analysis to detect undefined value uses in C programs
  - Inspired by our previous work (Sui *et al.* ISSTA ’12)
  - Guide instrumentation by solving a graph reachability problem

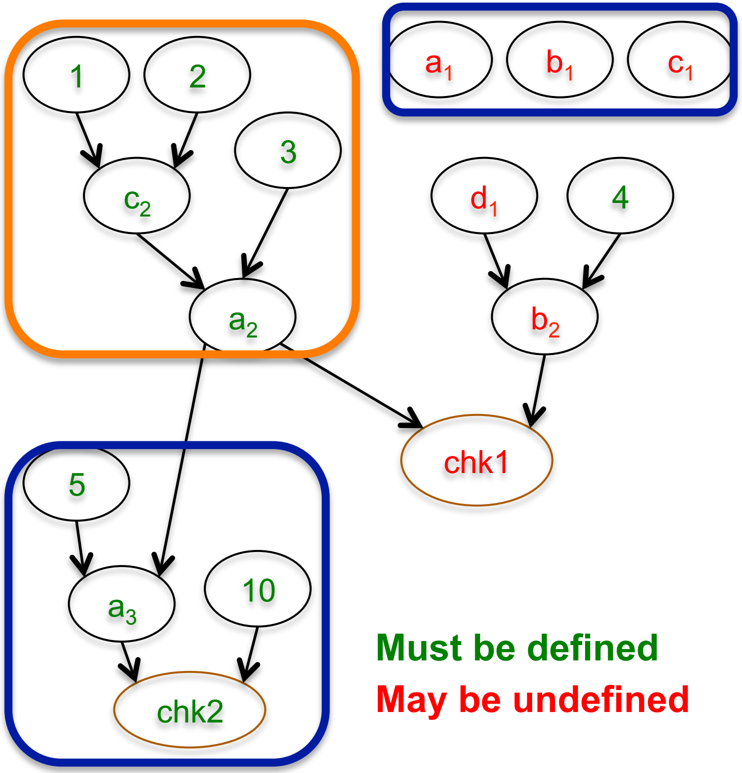
- Our value-flow representation allows optimizations to be developed to further reduce the instrumentation

- Usher reduces the slowdowns of MSan from 212% – 302% to 123% – 140% for 15 benchmarks

Outline

- What are undefined values?
- Related work and our contributions
- Methodology
- Evaluation
- Conclusion

A motivating example



Must be defined

May be undefined

void foo() {	a_s= F; b_s= F; c_s= F;
int a_1, b_1, c_1, d_1;	d_s= F;
$c_2 = 1 + 2$;	$c_s = T \& T$;
$a_2 = 3 + c_2$;	$a_s = T \& c_s$;
$b_2 = d_1 + 4$;	$b_s = d_s \& T$;
I1: if ($a_2 > b_2$) ...;	check($a_s \& b_s$);
...	
$a_3 = a_2 + 5$;	$a_s = a_s \& T$;
I2: if ($a_3 > 10$) ...;	check($a_s \& T$);
}	

Outline

- What are undefined values?
- Related work and our contributions
- Methodology
  - How to perform the static value-flow analysis?
  - How to guide the instrumentation?
  - How to further optimize the performance?
- Evaluation
- Conclusion

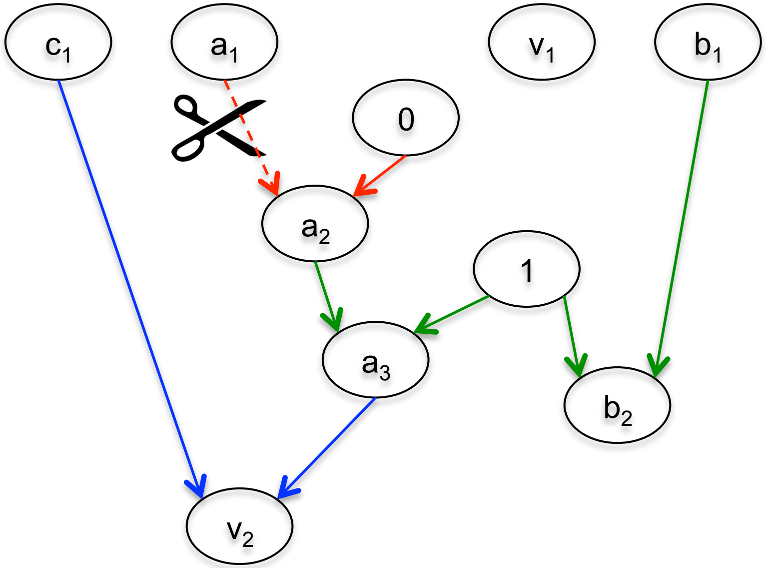
Value-flow graph (VFG)

- Sparseness is based on Static Single Assignment (SSA) form

- For top-level variables
  - e.g., x = y;, a = b + c;
  - SSA is straight forward (e.g. partial SSA in LLVM-IR)

- For address-taken variables
  - e.g., x = *p;,   *q = y;
  - Use pointer analysis results to build *Memory* SSA

VFG for address-taken variables



Points-to information:

$p_1 \rightarrow \{a\}$
$x_4 \rightarrow \{a, b\}$
$y_3 \rightarrow \{a, c\}$

```
void bar() {
    int a1, b1, c1, v1;
    int *p1, *x1, *y1;
    ...
   *p1 = 0;     [a2 = 0;]  //strong update
    ...
   *x4 = 1;     [a3=(1, a2);]  //weak update
                [b2=(1, b1);]
    ...
   v2 = *y3;    [v2=(a3, c1);]
}
```

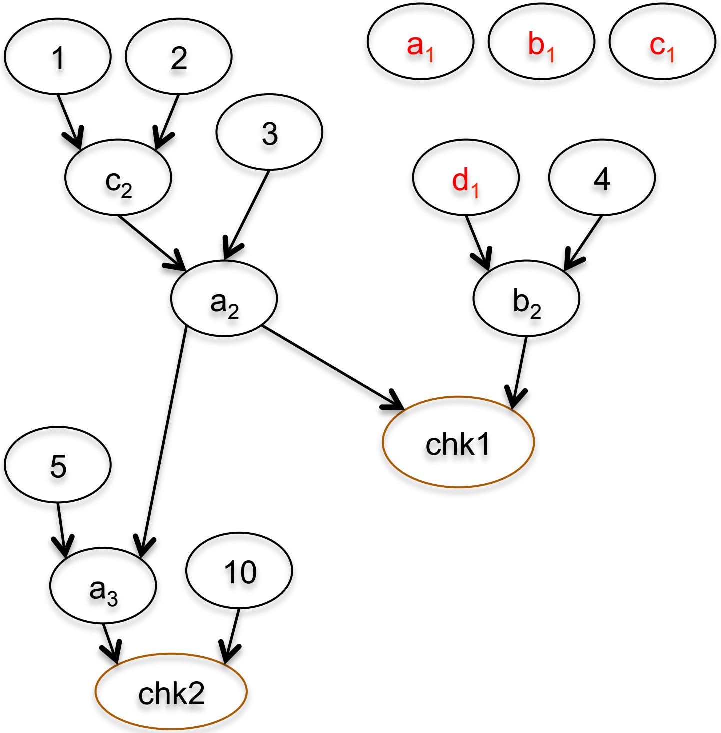
Outline

- What are undefined values?
- Related work and our contributions
- Methodology
  - How to perform the static value-flow analysis?
  - How to guide the instrumentation?
  - How to further optimize the performance?
- Evaluation
- Conclusion

Guided instrumentation

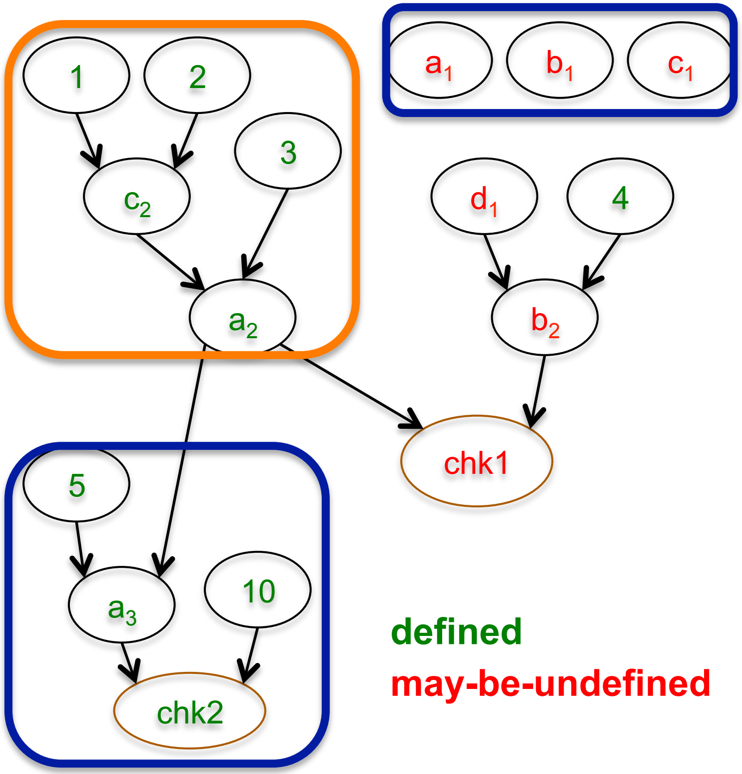
- Definedness resolution on VFG
  - Traverse from every undefined source node (stack, malloc())
    - Mark the reachable nodes as may-be-undefined
    - Mark the unreachable nodes as defined
  - Graph reachability in a context-sensitive manner
- Instrumentation
  - Rule out the nodes that never reach any may-be-undefined check node (critical operation)
  - For the remaining nodes
    - For may-be-undefined nodes, insert instrumentation code
    - For defined nodes,

Revisit the motivating example



void foo() {	a_s= F; b_s= F; c_s= F;
int a_1, b_1, c_1, d_1;	d_s= F;
$c_2 = 1 + 2$;	$c_s = T \& T$;
$a_2 = 3 + c_2$;	$a_s = T \& c_s$;
$b_2 = d_1 + 4$;	$b_s = d_s \& T$;
l1: if ($a_2 > b_2$) ...;	check($a_s \& b_s$);
...	
$a_3 = a_2 + 5$;	$a_s = a_s \& T$;
l2: if ($a_3 > 10$) ...;	check($a_s \& T$);
}	

Revisit the motivating example

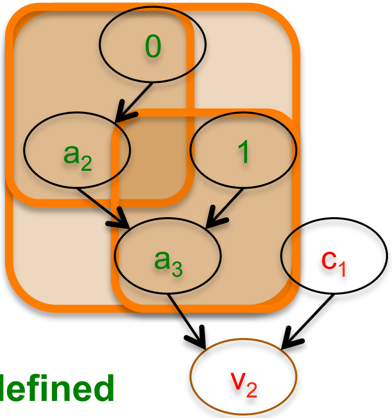


defined

may-be-undefined

void foo() {	a_s= F; b_s= F; c_s= F;
int a_1, b_1, c_1, d_1;	d_s= F;
c_2 = 1 + 2;	c_s = T & T;
a_2 = 3 + c_2;	a_s = T & c_s;
b_2 = d_1 + 4;	b_s = d_s & T;
l1: if (a_2 > b_2) ...;	check(a_s & b_s);
...	
a_3 = a_2 + 5;	a_s = a_s & T;
l2: if (a_3 > 10) ...;	check(a_s & T);
}	

False positives with address-taken variables



defined

may-be-undefined

Points-to information

- **Statically**
  - $p_1 \rightarrow \{a\}$
  - $x_4 \rightarrow \{a, b\}$
  - $y_3 \rightarrow \{a, c\}$
- **At runtime**
  - $p_1 \rightarrow a$
  - $x_4 \rightarrow b$
  - $y_3 \rightarrow a$

void bar() {	
int a_1, b_1, c_1, v_1;	a_s= F; b_s= F;
int $*p_1$, $*x_1$, $*y_1$;	c_s= F; v_s= F;
...	...
I1: $*p_1$ = 0; [a_2 = 0;] //SU	$(*p)_s$ = T;
...	
I2: $*x_4$ = 1; [a_3=(1, a_2);] //WU	$(*x)_s$ = T;
[b_2=(1, b_1);]	
...	...
I3: v_2 = $*y_3$; [v_2=(a_3, c_1);]	v_s = $(*y)_s$;
I4: if (v_2) ...;	check (v_s);
}	

Exe	a_s / b_s / v_s	a_s / b_s / v_s	a_s / b_s / v_s
I1	F / F / F	F / F / F	T / F / F
I2	F / F / F	F / T / F	T / F / F
I3	F / F / F	F / T / F	T / F / T
I4	F / F / **F**	F / T / **F**	T / F / **T**

Outline

- What are undefined values?
- Related work and our contributions
- Methodology
  - How to perform the static value-flow analysis?
  - How to guide the instrumentation?
  - How to further optimize the performance?
- Evaluation
- Conclusion

Optimizations on the VFG

- VFG simplification
  - Reduce shadow propagation distance

- Redundant check elimination
  - If a value x_n is previously checked, then the following checks on it can be eliminated

Redundant check elimination

...

$c_1 = a_1 + b_1$;

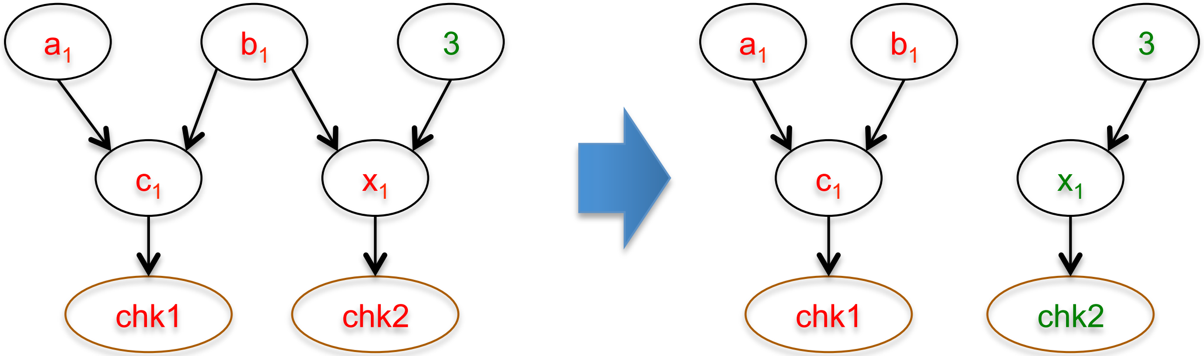
I1: if (c_1) ...

...

I2: $x_1 = b_1 + 3$;

if (x_1) ....

...



(1) Value V must flow to a checking statement at L;

(2) Value V is used in statement at L';

(3) L dominates L' in CFG.

If (1), (2) and (3) hold, cut the edge from V for L'

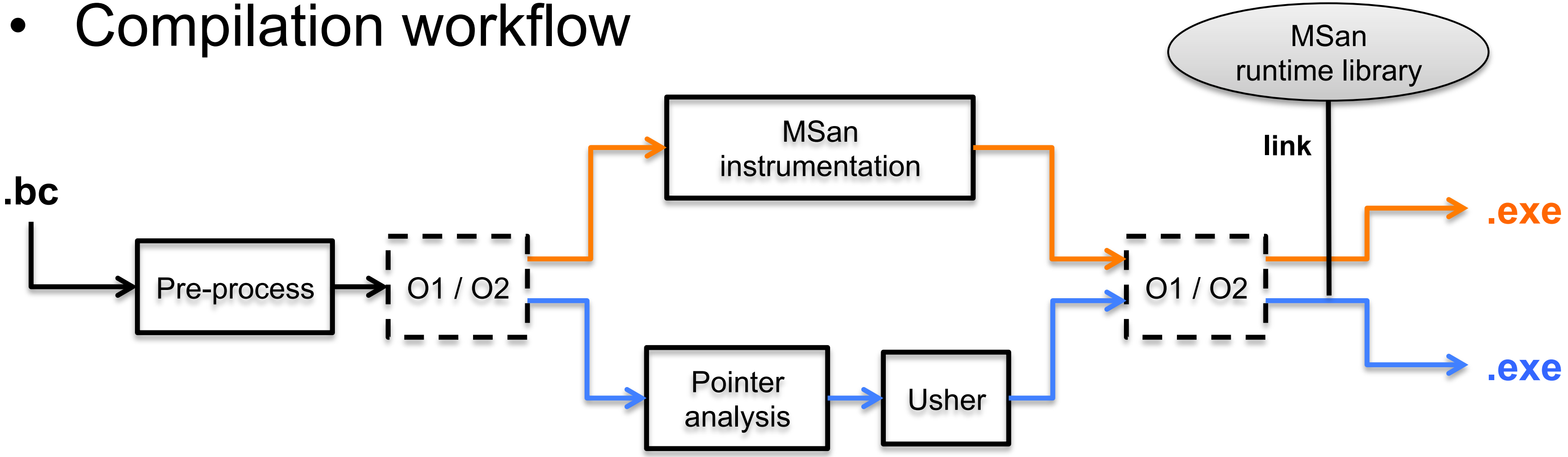
Outline

- Introduction of CORG@UNSW
- What are undefined values?
- Our solution
- Evaluation
- Conclusion

Evaluation

- Benchmarks
  - All 15 C programs of SPEC2K

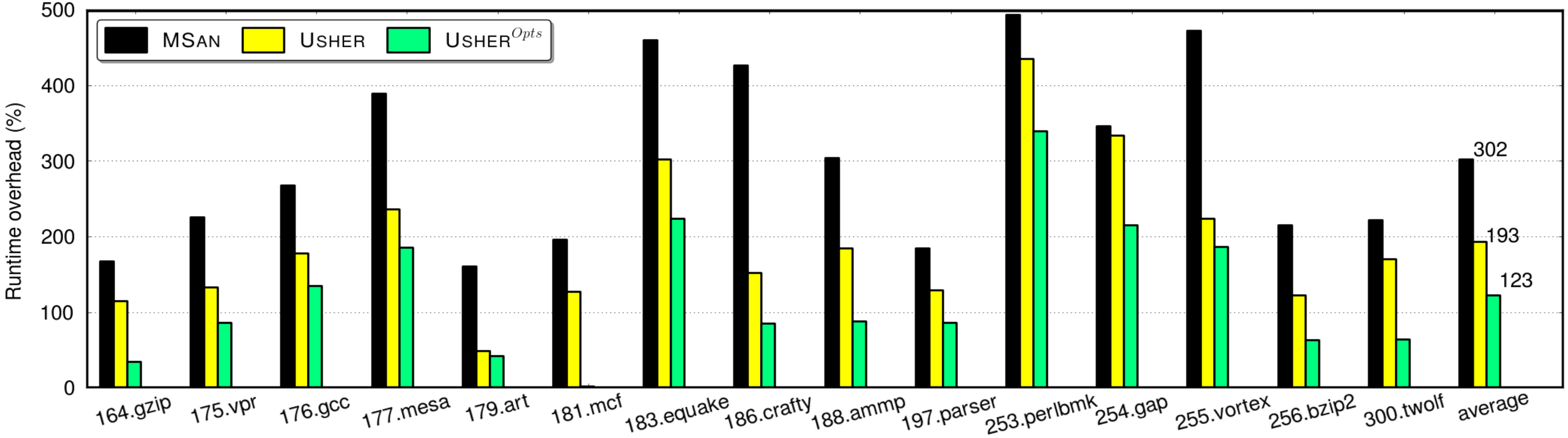
- Compilation workflow



Field-sensitive Andersen's analysis (Hardekopf and Lin, PLDI '07)

Results

- Static analysis
  - Most benchmarks <1s and <320MB
  - *176.gcc* (58s, 2.7GB) and *253.perlbmk* (54s, 1.4GB)
- Runtime overhead (WRT native code)



Outline

- Introduction of CORG@UNSW
- What are undefined values?
- Our solution
- Evaluation
- Conclusion

Conclusion

- A new static + dynamic analysis for undefined value use detection in C programs
  - Sparse VFG analysis
  - VFG-based optimizations
  - Selective instrumentation
- For even better results?
  - Try more precise pointer analysis

Thank You!