Automatically Generating Features for Learning Program Analysis Heuristics for C-Like Languages

Kwonsoo Chae
Korea University

Joint work with Hakjoo Oh (Korea University), Kihong Heo (Seoul National University), Hongseok Yang (University of Oxford)

OOPSLA'17 @Vancouver, Canada
Static Analysis

- Diverse engineering decisions in static analysis:
  - Context-sensitivity for which procedures?
  - Relational analysis for which variables?
  - Unsoundness for which part of the program?
  - etc.
Static Analysis

Data-Driven

• Diverse engineering decisions in static analysis:
  • Context-sensitivity for which procedures?
  • Relational analysis for which variables?
  • Unsoundness for which part of the program?
  • etc.

• Data-driven static analysis aims at automatically learning analysis heuristics from the codebase.
Main Obstacle:
Manual Feature Engineering

What people expect

codebase → ML → analysis
Main Obstacle: Manual Feature Engineering

What people expect

- codebase
- ML
- analysis

Reality

- codebase
- feature engineering
- ML
- analysis

❌ Manual, time-consuming
❌ Need for domain expertise
❌ Not interchangeable among different analyses
Main Obstacle: Feature Engineering

Reality

- Manual, time-consuming
- Need for domain expertise
- Not interchangeable among different analyses
Main Obstacle: Manual Feature Engineering

Reality

- Manual, time-consuming
- Need for domain expertise
- Not interchangeable among different analyses
Main Obstacle: Feature Engineering

- Manual, time-consuming
- Need for domain expertise
- Not interchangeable among different analyses
<table>
<thead>
<tr>
<th>Type</th>
<th>#</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>local variable</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>global variable</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>3-way relation</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>4-way relation</td>
</tr>
</tbody>
</table>

**Main Obstacle:**

Feature Engineering is manual, time-consuming, and requires domain expertise. Features are not interchangeable among different analyses.

---

### Manual Feature Engineering

- **Manual, time-consuming**
- **Need for domain expertise**
- **Not interchangeable among different analyses**

---

### What People Think vs. Reality

**What People Think**

**Reality**

- Manual Feature Engineering

---

### Example Feature

- **Description:**
  - $f_1(P, (x, y))$: $k$ represents a constant.
  - $P$ contains an assignment $x = y + k$ or $y = x + k$.
  - $P$ contains a guard $x \leq y + k$ or $y \leq x + k$.
  - $P$ contains a `malloc` of the form $x = \text{malloc}(y)$ or $y = \text{malloc}(x)$.
  - $P$ contains a command $x = \text{strlen}(y)$ or $y = \text{strlen}(x)$.
  - $P$ contains an `if` statement on the form $x[y] = y[x]$.
  - $P$ contains an expression that multiplies $x$ or $y$ by a constant different from 1.
  - $P$ contains an expression that divides $x$ or $y$ by a variable.
  - $P$ contains an expression that has $x$ or $y$ as an operand of bitwise operations.
  - $P$ contains an assignment that updates $x$ or $y$ using non-Octagonal expressions.
  - $x$ and $y$ are has the same name in different scopes.
  - $x$ and $y$ are both global variables in $P$.
  - $x$ or $y$ is a global variable in $P$.
  - $x$ or $y$ is a field of a structure in $P$.
  - $x$ and $y$ represent sizes of some arrays in $P$.
  - $x$ and $y$ are temporary variables in $P$.
  - $x$ or $y$ is a local variable of a recursive function in $P$.
  - $x$ or $y$ is tested for the equality with $\pm 1$ in $P$.
  - $x$ and $y$ represent sizes of some global arrays in $P$.
  - $x$ or $y$ stores the result of a library call in $P$.
  - $x$ and $y$ are local variables of different functions in $P$.
  - $(x, y)$ consists of a local var. and the size of a local array in different fun. in $P$.
  - $(x, y)$ consists of a local var. and a temporary var. in different functions in $P$.
  - $(x, y)$ consists of a global var. and the size of a local array in $P$.
  - $(x, y)$ contains a temporary var. and the size of a local array in $P$.
  - $(x, y)$ consists of local and global variables not accessed by the same fun. in $P$.
  - $x$ or $y$ is a self-updating global var. in $P$.
  - The flow-insensitive analysis of $P$ results in a finite interval for $x$ or $y$.
  - $x$ or $y$ is the size of a constant string in $P$. 

---

### Relations in SAS'16

- $\forall (x, y) \in P$, $\exists A \in \{ \text{local var}, \text{global var} \}$, $A \subseteq \{ x, y \}$.
Main Obstacle: Manual Feature Engineering

<table>
<thead>
<tr>
<th>Target</th>
<th>Feature</th>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>Feature</td>
<td>Property</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>Loop</td>
<td>Loop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Library</td>
<td>Library</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Main Obstacle:** Manual Feature Engineering

- **What people think**
- **Reality**

- **Main Obstacle:** Manual Feature Engineering

- **Loop**

- **Library**

- **Review:**

- **ICSE'17**

- **unsoundness**
Main Obstacle: Feature Engineering

What people think

- Manual Feature Engineering
- Context-sensitivity

Reality

- Main Obstacle: Manual Feature Engineering
## Main Obstacle: Feature Engineering

**What people think**

- **Reality**
  - **Main Obstacle:** Manual Feature Engineering

### Context-Sensitivity (previous talk)

- **Feature engineering:** major bottleneck in data-driven static analysis

### Table

<table>
<thead>
<tr>
<th>Type</th>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>leaf function</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>function containing malloc</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td>function containing realloc</td>
</tr>
<tr>
<td>A</td>
<td>4</td>
<td>function containing return</td>
</tr>
<tr>
<td>A</td>
<td>5</td>
<td>function containing #</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
<td>function containing 1</td>
</tr>
<tr>
<td>A</td>
<td>7</td>
<td>function containing (a = return 1)</td>
</tr>
<tr>
<td>A</td>
<td>8</td>
<td>function containing 3</td>
</tr>
<tr>
<td>A</td>
<td>9</td>
<td>function containing #</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>function containing 2</td>
</tr>
<tr>
<td>A</td>
<td>11</td>
<td>function containing 4</td>
</tr>
<tr>
<td>A</td>
<td>12</td>
<td>function containing 1</td>
</tr>
<tr>
<td>A</td>
<td>13</td>
<td>function containing 6</td>
</tr>
<tr>
<td>A</td>
<td>14</td>
<td>function containing 7</td>
</tr>
<tr>
<td>A</td>
<td>15</td>
<td>function containing 8</td>
</tr>
<tr>
<td>A</td>
<td>16</td>
<td>function containing 14</td>
</tr>
<tr>
<td>A</td>
<td>17</td>
<td>function containing 20</td>
</tr>
<tr>
<td>A</td>
<td>18</td>
<td>function containing 25</td>
</tr>
<tr>
<td>A</td>
<td>19</td>
<td>function containing 28</td>
</tr>
<tr>
<td>A</td>
<td>20</td>
<td>function containing 29</td>
</tr>
<tr>
<td>A</td>
<td>21</td>
<td>function containing 31</td>
</tr>
<tr>
<td>A</td>
<td>22</td>
<td>function containing 32</td>
</tr>
<tr>
<td>A</td>
<td>23</td>
<td>function containing 33</td>
</tr>
<tr>
<td>A</td>
<td>24</td>
<td>function containing 34</td>
</tr>
<tr>
<td>A</td>
<td>25</td>
<td>function containing 35</td>
</tr>
<tr>
<td>A</td>
<td>26</td>
<td>function containing 36</td>
</tr>
<tr>
<td>A</td>
<td>27</td>
<td>function containing 37</td>
</tr>
<tr>
<td>A</td>
<td>28</td>
<td>function containing 38</td>
</tr>
</tbody>
</table>

### Type of Features

<table>
<thead>
<tr>
<th>Target</th>
<th>Feature</th>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mall</td>
<td>Semantic</td>
<td>Numeric</td>
<td>Bool</td>
<td>Whether the loop condition contains null or not</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether the loop condition is determined by external inputs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether global variables are accessed in the loop condition</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether a variable has a finite interval value in the loop condition</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether a variable has a finite size of array in the loop condition</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether a variable has a finite string in the loop condition</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether a variable has an array of which the size is a left-closed interval</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether a variable has an array of which the offset is a left-closed interval</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The (normalized) number of abstract locations accessed in the loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether the parameters contain constants or not</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether the return type is void or not</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether the return type is int or not</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether the function is declared in string.h or not</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether the function is called in a loop or not</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The (normalized) number of arguments</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether a parameter are defined in a loop or not</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether the return value is used in a loop or not</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether a parameter is update via the library call</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether a parameters points to a global variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether parameters are determined by external inputs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Whether a parameter have a finite interval value</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The (normalized) number of abstract locations accessed in the arguments</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The (normalized) number of string arguments</td>
</tr>
</tbody>
</table>
We Aim At Generating Features Automatically

What people expect

- codebase
  - ML
    - analysis

Reality

- codebase
  - feature engineering
    - ML
      - analysis

Our goal

- codebase
  - automatically generated features
    - ML
      - analysis

This talk
Example: Partially Flow-Sensitive Interval Analysis

\[
x = 0; y = 0
\]

\[
z = \text{input()}
\]

\[
w = 0
\]

\[
y = x
\]

\[
y ++
\]

assert (y>0); assert (z > 0); assert (w ==0)
Example: Partially Flow-Sensitive Interval Analysis

```
x = 0; y = 0
z = input()
w = 0
y = x
y ++
assert (y>0); assert (z > 0); assert (w ==0)

✓ FS-proven but FI-unproven (FS is beneficial)
```
Example: Partially Flow-Sensitive Interval Analysis

\[
x = 0; \ y = 0
\]

\[
z = \text{input()}
\]

\[
w = 0
\]

\[
y = x
\]

\[
y ++
\]

\[
\text{assert (y>0); assert (z > 0); assert (w ==0)}
\]

✓ Unproven even by FS
(FS is not beneficial)
Example: Partially Flow-Sensitive Interval Analysis

```
x = 0; y = 0

z = input()

w = 0

y = x

y ++
```

```
assert (y>0); assert (z > 0); assert (w ==0)
```

✓ Proven even by FI
(FS is *not* beneficial)
Example: Partially Flow-Sensitive Interval Analysis

```
x = 0; y = 0
```

```
z = input()
```

```
w = 0
```

```
y = x
```

```
y ++
```

```
assert (y>0); assert (z > 0); assert (w ==0)
```
Example: Partially Flow-Sensitive Interval Analysis

```
x = 0; y = 0
```

```
z = input()
```

```
w = 0
```

```
y = x
```

```
y ++
```

- Build a "classifier" that selects the 1st assertion only.

```
assert (y>0);
assert (z > 0);
assert (w ==0)
```
Building a Classifier

• Usual procedure
  
  (1) Design a good set of features manually.

\[ F = \{ f_1, \ldots, f_k \} \]
Building a Classifier

- Usual procedure
  1. Design a good set of features manually.
  2. Generate labeled data.

\[
F = \{ f_1, \ldots, f_k \} \quad \Rightarrow \quad [f_1(Q_1), \ldots, f_k(Q_1)] : 1 \quad [f_1(Q_2), \ldots, f_k(Q_2)] : 0 \quad \ldots
\]
Building a Classifier

- Usual procedure
  1. Design a good set of features manually.
  2. Generate labeled data.
  3. Run an off-the-shelf classification algorithm.

\[ F = \{f_1, \ldots, f_k\} \Rightarrow \begin{cases} [f_1(Q_1), \ldots, f_k(Q_1)] : 1 \\ [f_1(Q_2), \ldots, f_k(Q_2)] : 0 \\ \vdots \end{cases} \Rightarrow \text{classification algorithm} \downarrow \text{classifier} \]
Building a Classifier

Our

Usual procedure

(1) Design a good set of features manually.

(2) Generate labeled data.

(3) Run an off-the-shelf classification algorithm.

\[ F = \{ f_1, \ldots, f_k \} \quad \Rightarrow \quad [f_1(Q_1), \ldots, f_k(Q_1)] : 1 \quad \Rightarrow \quad \text{classification algorithm} \]

\[ [f_1(Q_2), \ldots, f_k(Q_2)] : 0 \]

\[ \ldots \]

“automatically”
Highlight: Key Ideas

1. Capture the key reason why FS is beneficial using a program reducer.

```
for (i=0; i<7; i++)
assert (i<10);
```
Highlight: Key Ideas

1. Capture the key reason why FS is beneficial using a program reducer.

2. **Generalize** the reduced program.

```
for (i=0; i<7; i++)
    assert (i<10);
```

"FS ✔ & FI ✗"
1. Capture the key reason why FS is beneficial using a program reducer.

2. **Generalize** the reduced program.

```
for (i=0; i<7; i++)
assert (i<10);
```

-The program has the feature

new program

feature
Highlight: Results

- Generated 38 (interval), 45 (pointer), 44 (Octagon) features.
- Analysis heuristics built on top of automatically generated features.
- Excellent balance between cost and precision, e.g.,
  - Partially flow-sensitive interval analysis:

**Precision**

- FI (0)
- FS (100) 80.2%

**Cost**

- FI (1x) 2.0x
- FS (46x)
Automatic Feature Generation

Recipe

(1) Capture the key reasons from the codebase

(using a program reducer).

(2) Properly generalize the key reasons

(to build generic features).

The end.
(1) Capture The Key Reasons

```plaintext
1  a = 0; b = 0;
2  while (1) {
3      b = unknown();
4      if (a > b)
5          if (a < 3)
6              assert (a < 5);
7          a++;  
8  }
```
(1) **Capture The Key Reasons**

- Use a program reducer to generate a feature program.
- The reduction preserves an invariant $\phi$:

$$\phi(p, q) \equiv FI(p, q) = \text{unproven} \land FS(p, q) = \text{proven}$$

```plaintext
1  a = 0; b = 0;
2  while (1) {
3      b = unknown();
4      if (a > b)
5          if (a < 3)
6              assert (a < 5);
7              a++;
8  }
```

(reduce)

```plaintext
1  a = 0;
2  while (1) {
3      if (a < 3)
4          assert (a < 5);
5          a++;
6  }
```

(FS: “$a < 3$ before assertion in the loop”)

e.g., C-Reduce
(2) Generalize The Key Reasons

```
a = 0;
while (1) {
    if (a < 3)
        assert (a < 5);
    a++;
}
```
(2) Generalize The Key Reasons

- Properly generalize the feature program to an abstract data-flow graph (= feature).
(2) Generalize The Key Reasons

- Properly generalize the feature program to an abstract data-flow graph (= feature).
- The right level of abstraction is automatically identified by an iterative search and cross validation.

Generalization vs. Preservation
Feature Check = Graph Inclusion Check

**Feature:**

- \( x := x + c \)
- \( x := c \)
- \( x < c \)
- \( Q(x < c) \)

**Original Program:**

```plaintext
1  a = 0; b = 0;
2  while (1) {
3      b = unknown();
4      if (a > b)
5          if (a < 3)
6              assert (a < 5);
7          a++;
8  }
```
Feature Check = Graph Inclusion Check

feature:

original program:

```plaintext
a = 0; b = 0;
while (1) {
  b = unknown();
  if (a > b)
    if (a < 3)
      assert (a < 5);
  a++;
}
```

abstract data-flow graph:
Feature Check = Graph Inclusion Check

feature:

original program:

```
1  a = 0; b = 0;
2  while (1) {
3      b = unknown();
4      if (a > b)
5          if (a < 3)
6              assert (a < 5);
7          a++;
8  }
```

abstract data-flow graph:
Feature Check = Graph Inclusion Check

feature:

original program:

1 a = 0; b = 0;
2 while (1) {
3 b = unknown();
4 if (a > b)
5 if (a < 3)
6 assert (a < 5);
7 a++;
8 }

abstract data-flow graph:

"noise"
Feature Check = Graph Inclusion Check

original program:
1  a = 0; b = 0;
2  while (1) {
3      b = unknown();
4      if (a > b)
5          if (a < 3)
6              assert (a < 5);
7      a++;    
8  }

abstract data-flow graph:

transitive closure

feature:
Feature Check = Graph Inclusion Check

original program:
1  a = 0; b = 0;
2  while (1) {
3      b = unknown();
4      if (a > b)
5          if (a < 3)
6              assert (a < 5);
7          a++;
8  }

abstract data-flow graph:

Removing noise:
✔ reducer (offline, feature)
✔ transitive closure (online, new pgm)

- feature:
Feature Check = Graph Inclusion Check

original program:
1  a = 0; b = 0;
2  while (1) {
3    b = unknown();
4    if (a > b)
5      if (a < 3)
6        assert (a < 5);
7      a++;
8  }

abstract data-flow graph:
Evaluation

- Static analyzer: Sparrow (https://github.com/ropas/sparrow)
- Reducer: C-Reduce [PLDI'12] (https://embed.cs.utah.edu/creduce)
- Three instance analyses for C
  - Partially flow-sensitive interval analysis
  - Partially flow-sensitive pointer analysis
  - Partial Octagon analysis
- 60 benchmark programs from Linux and GNU packages
### Results: Effectiveness (Classifier)

#### Partially flow-sensitive interval analysis

<table>
<thead>
<tr>
<th>Trial</th>
<th>Query Prediction</th>
<th>#Proved Queries</th>
<th>Analysis Cost (sec)</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Prove</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
<td>Flr (a)</td>
<td>Fsr (b)</td>
</tr>
<tr>
<td>1</td>
<td>71.5 %</td>
<td>78.9 %</td>
<td>6,537</td>
<td>7126</td>
</tr>
<tr>
<td>2</td>
<td>60.9 %</td>
<td>75.1 %</td>
<td>4,127</td>
<td>4,544</td>
</tr>
<tr>
<td>3</td>
<td>78.3 %</td>
<td>74.0 %</td>
<td>6,701</td>
<td>7,532</td>
</tr>
<tr>
<td>4</td>
<td>73.0 %</td>
<td>76.2 %</td>
<td>4,399</td>
<td>4,956</td>
</tr>
<tr>
<td>5</td>
<td>83.2 %</td>
<td>75.4 %</td>
<td>5,676</td>
<td>6,277</td>
</tr>
<tr>
<td>TOTAL</td>
<td>74.5 %</td>
<td>75.8 %</td>
<td>27,440</td>
<td>30,435</td>
</tr>
</tbody>
</table>

#### Partially flow-sensitive pointer analysis

<table>
<thead>
<tr>
<th>Trial</th>
<th>Query Prediction</th>
<th>#Proved Queries</th>
<th>Analysis Cost (sec)</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Prove</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
<td>Flp</td>
<td>FSp</td>
</tr>
<tr>
<td>1</td>
<td>79.2 %</td>
<td>76.8 %</td>
<td>4,399</td>
<td>6,346</td>
</tr>
<tr>
<td>2</td>
<td>78.3 %</td>
<td>77.2 %</td>
<td>7,029</td>
<td>8,650</td>
</tr>
<tr>
<td>3</td>
<td>74.6 %</td>
<td>75.0 %</td>
<td>8,781</td>
<td>10,352</td>
</tr>
<tr>
<td>4</td>
<td>73.9 %</td>
<td>76.0 %</td>
<td>10,559</td>
<td>12,914</td>
</tr>
<tr>
<td>5</td>
<td>78.0 %</td>
<td>82.5 %</td>
<td>4,205</td>
<td>5,705</td>
</tr>
<tr>
<td>TOTAL</td>
<td>76.7 %</td>
<td>77.4 %</td>
<td>34,973</td>
<td>43,967</td>
</tr>
</tbody>
</table>

#### Partial Octagon analysis

<table>
<thead>
<tr>
<th>Trial</th>
<th>Query Prediction</th>
<th>#Proved Queries</th>
<th>Analysis Cost (sec)</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Prove</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
<td>FSr</td>
<td>IMPCT</td>
</tr>
<tr>
<td>1</td>
<td>74.8 %</td>
<td>81.3 %</td>
<td>3,678</td>
<td>3,806</td>
</tr>
<tr>
<td>2</td>
<td>84.1 %</td>
<td>82.6 %</td>
<td>5,845</td>
<td>6,004</td>
</tr>
<tr>
<td>3</td>
<td>82.8 %</td>
<td>73.0 %</td>
<td>1,926</td>
<td>2,079</td>
</tr>
<tr>
<td>4</td>
<td>77.6 %</td>
<td>85.2 %</td>
<td>2,221</td>
<td>2,335</td>
</tr>
<tr>
<td>5</td>
<td>71.6 %</td>
<td>78.4 %</td>
<td>2,886</td>
<td>2,962</td>
</tr>
<tr>
<td>TOTAL</td>
<td>79.0 %</td>
<td>79.9 %</td>
<td>16,556</td>
<td>17,186</td>
</tr>
</tbody>
</table>
Results: Effectiveness \textbf{(Analysis)}

- Partially flow-sensitive interval analysis

<table>
<thead>
<tr>
<th>Trial</th>
<th>Query Prediction</th>
<th>#Proved Queries</th>
<th>Analysis Cost (sec)</th>
<th>Quality</th>
<th>(Oh et al. 2015)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
<td>FIr (a)</td>
<td>FSt (b)</td>
<td>Ours (c)</td>
</tr>
<tr>
<td>1</td>
<td>71.5%</td>
<td>78.9%</td>
<td>6,537</td>
<td>7,126</td>
<td>7,019</td>
</tr>
<tr>
<td>2</td>
<td>60.9%</td>
<td>75.1%</td>
<td>4,127</td>
<td>4,544</td>
<td>4,487</td>
</tr>
<tr>
<td>3</td>
<td>78.3%</td>
<td>74.0%</td>
<td>6,701</td>
<td>7,532</td>
<td>7,337</td>
</tr>
<tr>
<td>4</td>
<td>73.0%</td>
<td>76.2%</td>
<td>4,399</td>
<td>4,956</td>
<td>4,859</td>
</tr>
<tr>
<td>5</td>
<td>83.2%</td>
<td>75.4%</td>
<td>5,676</td>
<td>6,277</td>
<td>6,140</td>
</tr>
<tr>
<td>TOTAL</td>
<td>74.5%</td>
<td>75.8%</td>
<td>27,440</td>
<td>30,435</td>
<td>29,842</td>
</tr>
</tbody>
</table>

- Partially flow-sensitive pointer analysis

<table>
<thead>
<tr>
<th>Trial</th>
<th>Query Prediction</th>
<th>#Proved Queries</th>
<th>Analysis Cost (sec)</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
<td>FIr</td>
<td>FSp</td>
</tr>
<tr>
<td>1</td>
<td>79.2%</td>
<td>76.8%</td>
<td>4,399</td>
<td>6,346</td>
</tr>
<tr>
<td>2</td>
<td>78.3%</td>
<td>77.2%</td>
<td>7,029</td>
<td>8,650</td>
</tr>
<tr>
<td>3</td>
<td>74.6%</td>
<td>75.0%</td>
<td>8,781</td>
<td>10,352</td>
</tr>
<tr>
<td>4</td>
<td>73.9%</td>
<td>76.0%</td>
<td>10,559</td>
<td>12,914</td>
</tr>
<tr>
<td>5</td>
<td>78.0%</td>
<td>82.5%</td>
<td>4,205</td>
<td>5,705</td>
</tr>
<tr>
<td>TOTAL</td>
<td>76.7%</td>
<td>77.4%</td>
<td>34,973</td>
<td>43,967</td>
</tr>
</tbody>
</table>

- Partial Octagon analysis

<table>
<thead>
<tr>
<th>Trial</th>
<th>Query Prediction</th>
<th>#Proved Queries</th>
<th>Analysis Cost (sec)</th>
<th>Quality</th>
<th>(Heo et al. 2016)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
<td>FSt</td>
<td>IMPCT</td>
<td>Ours</td>
</tr>
<tr>
<td>1</td>
<td>74.8%</td>
<td>81.3%</td>
<td>3,678</td>
<td>3,806</td>
<td>3,789</td>
</tr>
<tr>
<td>2</td>
<td>84.1%</td>
<td>82.6%</td>
<td>5,845</td>
<td>6,004</td>
<td>5,977</td>
</tr>
<tr>
<td>3</td>
<td>82.8%</td>
<td>73.0%</td>
<td>1,926</td>
<td>2,079</td>
<td>2,036</td>
</tr>
<tr>
<td>4</td>
<td>77.6%</td>
<td>85.2%</td>
<td>2,221</td>
<td>2,335</td>
<td>2,313</td>
</tr>
<tr>
<td>5</td>
<td>71.6%</td>
<td>78.4%</td>
<td>2,886</td>
<td>2,962</td>
<td>2,944</td>
</tr>
<tr>
<td>TOTAL</td>
<td>79.0%</td>
<td>79.9%</td>
<td>16,556</td>
<td>17,186</td>
<td>17,067</td>
</tr>
</tbody>
</table>
Results: Comparison

- Partially flow-sensitive interval analysis

<table>
<thead>
<tr>
<th>Trial</th>
<th>Query Prediction</th>
<th>#Proved Queries</th>
<th>Analysis Cost (sec)</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
<td>FIr (a)</td>
<td>FSr (b)</td>
</tr>
<tr>
<td>1</td>
<td>71.5 %</td>
<td>78.9 %</td>
<td>6,537</td>
<td>7,126</td>
</tr>
<tr>
<td>2</td>
<td>60.9 %</td>
<td>75.1 %</td>
<td>4,127</td>
<td>4,544</td>
</tr>
<tr>
<td>3</td>
<td>78.3 %</td>
<td>74.0 %</td>
<td>6,701</td>
<td>7,532</td>
</tr>
<tr>
<td>4</td>
<td>73.0 %</td>
<td>76.2 %</td>
<td>4,399</td>
<td>4,956</td>
</tr>
<tr>
<td>5</td>
<td>83.2 %</td>
<td>75.4 %</td>
<td>5,676</td>
<td>6,277</td>
</tr>
<tr>
<td>TOTAL</td>
<td>74.5 %</td>
<td>75.8 %</td>
<td>27,440</td>
<td>30,435</td>
</tr>
</tbody>
</table>

- Partial Octagon analysis

<table>
<thead>
<tr>
<th>Trial</th>
<th>Query Prediction</th>
<th>#Proved Queries</th>
<th>Analysis Cost (sec)</th>
<th>Quality</th>
<th>(Herold et al. 2015)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
<td>FIr</td>
<td>IMPCT</td>
<td>Ours</td>
</tr>
<tr>
<td>1</td>
<td>74.8 %</td>
<td>81.3 %</td>
<td>3,678</td>
<td>3,806</td>
<td>3,789</td>
</tr>
<tr>
<td>2</td>
<td>84.1 %</td>
<td>82.6 %</td>
<td>5,845</td>
<td>6,004</td>
<td>5,977</td>
</tr>
<tr>
<td>3</td>
<td>82.8 %</td>
<td>73.0 %</td>
<td>1,926</td>
<td>2,079</td>
<td>2,036</td>
</tr>
<tr>
<td>4</td>
<td>77.6 %</td>
<td>85.2 %</td>
<td>2,221</td>
<td>2,335</td>
<td>2,313</td>
</tr>
<tr>
<td>5</td>
<td>71.6 %</td>
<td>78.4 %</td>
<td>2,886</td>
<td>2,962</td>
<td>2,944</td>
</tr>
<tr>
<td>TOTAL</td>
<td>79.0 %</td>
<td>79.9 %</td>
<td>16,556</td>
<td>17,186</td>
<td>17,067</td>
</tr>
</tbody>
</table>

- Consistently perform well on a wide range of programs. (↔ wide variation)
- No clear conclusion (different approaches and learning algorithms)
Results: Generated Features (Top 2)

- Partially flow-sensitive interval analysis

feature program 1:

```c
int buf [10];
for (i = 0; i < 7; i++) {
    buf[i] = 0;  // Query
}
```

feature program 2:

```c
k = 255; p = malloc (k);
while (k > 0) {
    *(p + k) = 0;  // Query
    k--;  
}
```

- Access to a consecutive memory region in a loop
- Bounded indice by a constant
Results: Generated Features (Top 2)

- Partially flow-sensitive pointer analysis

**feature program 1:**

```c
int j = 16; q = malloc(j)
if (q == 0)
    return;
else *q = 0; // Query
```

**feature program 2:**

```c
r = malloc(v);
r = &a;
*r = 0;  // Query
```

- Null-check before buffer access
- Strong update by the address of another variable
Results: Generated Features (Top 2)

- Partial Octagon analysis

**feature program 1:**

```
size = POS_NUM;
arr = malloc(size);
arr[size-1] = 0; // Query
```

**feature program 2:**

```
idx = POS_NUM;
buf = malloc(idx);
for (n = 0; n < idx; n++) {
    buf[n] = 0; // Query
}
```

- Array of a **positive size**
  - e.g., when POS_NUM = [1, +\(\infty\)] in the flow-sensitive interval analysis
- Index related to the size **in a simple linear way**
Caveats: Expressiveness of Features

- Our feature representation is expressive enough, but not perfect, e.g.,
  √ “x and y results in finite intervals after analysis.” ✗
  √ “$2^k$ type of integers are important constants.” ✗
"Features" in data-driven static analysis

- By reducing programs
- As generalized graphs (↔ program text)