Data-Driven and Focused Program Analysis

Hakjoo Oh
Korea University

15 November 2018
Static Program Analysis

Technology for “software MRI”

- Predict software behavior statically and automatically
  - static: analyzing program text without execution
  - automatic: sw is analyzed by sw ("static analyzer")
- Next-generation software testing technology
  - finding bugs early / full automation / all bugs found
- Being widely used in sw industry
Long-Standing Open Problem

- How to achieve soundness, precision, and scalability at the same time?
Long-Standing Open Problem

• How to achieve soundness, precision, and scalability at the same time?

Soundness

Consider all execution states

Scalability

Precision
Long-Standing Open Problem

- How to achieve soundness, precision, and scalability at the same time?

- Soundness
  - Consider all execution states

- Scalability

- Precision
  - Report few false alarms
Long-Standing Open Problem

- How to achieve soundness, precision, and scalability at the same time?

Soundness → Consider all execution states

- Scalability → Scale to large programs
- Precision → Report few false alarms
Long-Standing Open Problem

- How to achieve soundness, precision, and scalability at the same time?

Soundness

Consider all execution states

Scalability

Scale to large programs

“bug-finders”

Precision

Report few false alarms
Long-Standing Open Problem

- How to achieve soundness, precision, and scalability at the same time?

![Diagram showing soundness, scalability, and precision]

- Soundness: Consider all execution states
- Scalability: Scale to large programs
- Precision: Report few false alarms

“program verifiers”
Long-Standing Open Problem

- How to achieve soundness, precision, and scalability at the same time?

“optimizing compilers”

- Soundness
  - Consider all execution states

- Scalability
  - Scale to large programs

- Precision
  - Report few false alarms
Project Goal

• General technology for achieving soundness, precision, and scalability:

• Prove generality & effectiveness with three analyses:
  • numeric analysis scalable to 1M in 1 hour
  • pointer analysis scalable to 500K in 1 hour
  • symbolic analysis scalable to 300K in 1 hour
Approach Overview

- **Selective** application of high precision (and soundness):
  - cheap but imprecise
  - precise but expensive
  - cheap and precise

- Data-driven, automatic generation of selection heuristics:
  - Heuristics for deciding when to apply high precision
  - machine learning for program analysis
Example: Context-Sensitivity

```c
int h(n) {ret n;}

void f(a) {
  x = h(a);
  assert(x > 0);  // Query holds always
  y = h(input());
}

c3: void g() {f(8);}

  void m() {
  c4:  f(4);
  c5:  g();
  c6:  g();
  }
```
Context-Insensitive Analysis

- Merge calling contexts into single abstract context

```c
int h(n) { ret n; }

void f(a) {
  c1:  x = h(a);
       assert(x > 0);
  c2:  y = h(input());
}

c3: void g() { f(8); }

void m() {
  c4:  f(4);
  c5:  g();
  c6:  g();
}
```

cheap but imprecise
**k-Context-Sensitive Analysis**

- Analyze functions separately for each calling context

```c
int h(n) { return n; }

void f(a) {
    x = h(a);
    assert(x > 0);
    y = h(input());
}

c3: void g() { f(8); }

void m() {
    f(4);
    g();
    g();
}
```

(k=3)

precise but expensive
Selective Context-Sensitivity

- Selectively differentiate contexts only when necessary

```c
int h(n) { ret n; }

void f(a) {
    c1: x = h(a);
    assert(x > 0);
    c2: y = h(input());
}

c3: void g() { f(8); }

c4: void m() {
    c4: f(4);
    c5: g();
    c6: g();
}
```

Apply 2-ctx-sens: \{h\}
Apply 1-ctx-sens: \{f\}
Apply 0-ctx-sens: \{g, m\}

cheap and precise
Selective Context-Sensitivity

- Selectively differentiate contexts only when necessary

```c
int h(n) {ret n;}
void f(a) {
  c1: x = h(a);
  assert(x > 0);
  c2: y = h(input);
}
void g() {f(8);}
void m() {
  c4: f(4);
  c5: g();
  c6: g();
}
```

Apply 2-ctx-sens: {h}
Apply 1-ctx-sens: {f}
Apply 0-ctx-sens: {g, m}

Challenge: how to design a good selection heuristic?

cheap and precise
Hard Search Problem

- Intractably large and sparse search space, if not infinite
  - e.g., $S^k$ choices where $S = 2^{|Proc|}$ for $k$-context-sensitivity
- Real programs are complex to reason about
  - e.g., typical call-graph of real program:

A fundamental problem in selective program analysis

=> New data-driven approach
Existing Approaches

- Selection heuristics manually crafted by analysis experts:
  - pre-analysis [PLDI’14a, PLDI’14b, OOPSLA’18c]
  - dynamic analysis [POPL’12]
  - online refinement [PLDI’14c, POPL’17]
- Our claim: manual approaches are inherently limited:
  - nontrivial, sub-optimal, and unstable

Our direction: automatically generate heuristics via learning
Direction and Achievement

• **Learning algorithms** for data-driven program analysis
  • learning models [OOPSLA’17a]
  • optimization algorithms [TOPLAS’19]
  • feature engineering [OOPSLA’17b]

• **State-of-the-art program analyses** enabled by algorithms
  • interval / pointer analysis [OOPSLA’18a, TOPLAS’18]
  • symbolic analysis / execution [ICSE’18, ASE’18]
  • others program analyses [FSE’18, OOPSLA’18b]

9 papers in top-tier PL/SE conferences and journals
Learning Algorithm Overview

Parametric program analyzer

Training data (programs w/o labels)

Atomic features (a1,a2,…,a25)

e.g., procedures have invocation stmt, procedures return strings, etc

Learning Algorithm

Learned heuristic for applying context-sensitivity:

f2: procedures to apply 2-context-sensitivity

1 ∧ ¬3 ∧ ¬6 ∧ 8 ∧ ¬9 ∧ ¬16 ∧ ¬17 ∧ ¬18 ∧ ¬19 ∧ ¬20 ∧ ¬21 ∧ ¬22 ∧ ¬23 ∧ ¬24 ∧ ¬25

f1: procedures to apply 1-context-sensitivity

(1 ∧ ¬3 ∧ ¬4 ∧ ¬7 ∧ ¬8 ∧ 6 ∧ ¬9 ∧ ¬15 ∧ ¬16 ∧ ¬17 ∧ ¬18 ∧ ¬19 ∧ ¬20 ∧ ¬21 ∧ ¬22 ∧ ¬23 ∧ ¬24 ∧ ¬25) ∨
(¬3 ∧ ¬4 ∧ ¬7 ∧ ¬8 ∧ ¬9 ∧ 10 ∧ 11 ∧ 12 ∧ 13 ∧ ¬16 ∧ ¬17 ∧ ¬18 ∧ ¬19 ∧ ¬20 ∧ ¬21 ∧ ¬22 ∧ ¬23 ∧ ¬24 ∧ ¬25) ∨
(¬3 ∧ ¬9 ∧ 13 ∧ 14 ∧ 15 ∧ ¬16 ∧ ¬17 ∧ ¬18 ∧ ¬19 ∧ ¬20 ∧ ¬21 ∧ ¬22 ∧ ¬23 ∧ ¬24 ∧ ¬25) ∨
(1 ∧ 2 ∧ ¬3 ∧ 4 ∧ ¬5 ∧ ¬6 ∧ ¬7 ∧ ¬8 ∧ ¬9 ∧ ¬10 ∧ ¬13 ∧ ¬15 ∧ ¬16 ∧ ¬17 ∧ ¬18 ∧ ¬19 ∧ ¬20 ∧ ¬21 ∧ ¬22 ∧ ¬23 ∧ ¬24 ∧ ¬25)
State-of-the-art Pointer Analysis

- Achieved state-of-the-art pointer analysis for Java
- Foundational static analysis for bug-finders, verifiers, etc
- Trained with 5 small programs from the DaCapo benchmark and tested with 5 remaining large programs

**ROC curves for different datasets**

- **bloat**
  - Analysis time
  - # of may-fail casts
- **jython**
  - Analysis time
  - # of may-fail casts
State-of-the-art Pointer Analysis

- Achieved state-of-the-art pointer analysis for Java
- Foundational static analysis for bug-finders, verifiers, etc
- Trained with 5 small programs from the DaCapo benchmark and tested with 5 remaining large programs
State-of-the-art Pointer Analysis

- Cracked down the precision limit of pointer analysis
- Key enablers:
  - keep most important $k$, rather than most recent $k$
  - data-driven method determines importance

![Diagram showing concrete and abstract values with k=3]

![Graph showing analysis time vs. number of may fail casts with k=1 and k=2, and applying tunneling]

Performance Highlight:
- Hybrid 1-object sensitivity with context tunneling is faster than k=1, and more precise than k=2
Application to Symbolic Execution

- Symbolic execution: program analysis popular for bug-finding
- Relies on **path-selection heuristics**, typically hand-tuned:
  - e.g., CGS [FSE’14], CarFast [FSE’12], CFDS [ASE’08], Generational [NDSS’08], DFS [PLDI’05], …

Our goal: automatically generating path-selection heuristics
**State-of-the-art Symbolic Execution**

- Developed “data-driven symbolic execution”
- Considerable increase in code coverage
- Dramatic increase in bug-finding capability

<table>
<thead>
<tr>
<th>Phenomenons</th>
<th>Bug-Triggering Inputs</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>sed</td>
<td>Memory Exhaustion</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>g</td>
<td>4.4(latest)</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>sed</td>
<td>Infinite File Write</td>
<td>w-[</td>
</tr>
<tr>
<td></td>
<td>x; D</td>
<td></td>
</tr>
<tr>
<td>grep</td>
<td>Segmentation Fault</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>?()</code>?D+``</td>
<td></td>
</tr>
<tr>
<td>grep</td>
<td>Non-Terminating</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>(?(&quot;+&quot;?)+\w[8957]?)</code></td>
<td></td>
</tr>
<tr>
<td>gawk</td>
<td>Memory Exhaustion</td>
<td><code>6672467e2-E7</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.2(latest)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>OURS</th>
<th>CFDS</th>
<th>CGS</th>
<th>Random</th>
<th>Gen</th>
<th>DFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>gawk-3.0.3</td>
<td>100/100</td>
<td>0/100</td>
<td>0/100</td>
<td>0/100</td>
<td>0/100</td>
<td>0/100</td>
</tr>
<tr>
<td>grep-2.2</td>
<td>47/100</td>
<td>0/100</td>
<td>5/100</td>
<td>0/100</td>
<td>0/100</td>
<td>0/100</td>
</tr>
</tbody>
</table>
Application to Program Repair

- Program analysis for automatically finding and fixing SW errors
- Manual debugging is time-consuming and error-prone
  - e.g., double-free error in Linux kernel:

```c
in = malloc(1);
out = malloc(1);
... // use in, out
free(out);
free(in);

in = malloc(2);
if (in == NULL) {
    goto err;
}
out = malloc(2);
if (out == NULL) {
    free(in);
    goto err;
}
... // use in, out
free(out);
free(in);

in = malloc(2);
in = malloc(2);
if (in == NULL) {
    out = NULL;
    goto err;
}
free(out);
out = malloc(2);
if (out == NULL) {
    free(in);
    goto err;
}
... // use in, out
free(out);
free(in);

err:
free(in);
free(out);
return;
```

**Our goal:** data-driven static analysis for program repair
State-of-the-art Program Repair

• Safe and sound repair technique for memory errors
  • no errors introduced, generated patches are correct
• Static analysis enabled by our data-driven approach played a key role

![Diagram showing static analysis and SAT solver processes]

```c
in = malloc(1);
out = malloc(1);
... // use in, out
free(out);
free(in);

in = malloc(2);
if (in == NULL) {
    goto err;
}
out = malloc(2);
if (out == NULL) {
    free(in);
    goto err;
}
... // use in, out
err:
    free(in);
    free(out);
    return;
```

```c
in = malloc(1);
out = malloc(1);
... // use in, out
free(out);
free(in);

in = malloc(2);
if (in == NULL) {
    goto err;
}
out = malloc(2);
if (out == NULL) {
    free(in);
    goto err;
}
... // use in, out
err:
    free(in);
    free(out);
    return;
```
Being recognized as a new and promising research direction

“Overall, the paper addresses an important problem advancing the state of the art in the **very interesting and promising area** of data-driven program analysis.” (from OOPSLA reviews)

“Supremely well-written and is clearly situated within related work, addressing a **clearly high-profile long-standing problem** of scalability.” (from OOPSLA reviews)

“I really enjoyed reading. It tackles a **fundamental problem** in program analysis – developing heuristics to localize precision, and presents a **very novel approach**.” (from TOPLAS reviews)

“There is a **novel idea**, which is not only **neat and elegant**, but I think may apply to other machine learning domains in program analysis.” (from OOPSLA reviews)

“The algorithm that automatically generates search heuristics for concolic testing is **novel and very interesting**.” (from ICSE reviews)
Future Research Directions

• Just started; Need to extend depth and breadth
• **Foundational algorithms** for data-driven program analysis
  • expressiveness, efficiency, generality, automation
  • unified and reusable framework
• **Applications** to various real problems
  • heuristics × analyses × languages
• **Deployment** in industrial tools
Summary

Our Data-Driven Program Analysis

- **Goal:** Achieving the ideal program analysis technology
- **Approach:** Data-driven program analysis
- **Research Directions:**
  - new and foundational algorithms
  - practical and diverse applications
- **Impacts:**
  - solving the longstanding open problem
  - paradigm-shift in program analysis research
Summary

Our Data-Driven Program Analysis

• **Goal**: Achieving the ideal program analysis technology

• **Approach**: Data-driven program analysis

• **Research Directions**:
  • new and foundational algorithms
  • practical and diverse applications

• **Impacts**:
  • solving the longstanding open problem
  • paradigm-shift in program analysis research

Thank you
References

Ours

• [PLDI’14a] Selective context-sensitivity guided by impact pre-analysis
• [OOPSLA’15] Learning a Strategy for Adapting a Program Analysis via Bayesian Optimisation
• [OOPSLA’17a] Data-driven context-sensitivity for points-to analysis
• [OOPSLA’17b] Automatically generating features for learning program analysis heuristics
• [OOPSLA’18a] Precise and scalable points-to analysis via data-driven context tunneling
• [OOPSLA’18b] Automatic diagnosis and correction of logical errors for functional programming assignments
• [ICSE’18] Automatically generating search heuristics for concolic testing
• [FSE’18] MemFix: Static-analysis-based repair of memory deallocation errors for C
• [ASE’18] Template-guided concolic testing via online learning
• [TOPLAS’18] Adaptive static analysis via learning with bayesian optimization
• [TOPLAS’19] A machine-learning algorithm with disjunctive model for data-driven program analysis

Others

• [PLDI’14b] Introspective analysis: context-sensitivity, across the board
• [PLDI’14c] On abstraction refinement for program analyses in Datalog
• [POPL’12] Abstractions from tests
• [POPL,17] Semantic-directed clumping of disjunctive abstract states
• [OOPSLA’18c] Precision-guided context-sensitivity for pointer analysis
Learning Algorithm Detail

• Each sub-heuristic \( f_i \) is a boolean combination of features
  \[
f \rightarrow \text{true}|\text{false}|a_i \in \mathbb{A} | \neg f | f_1 \land f_2 | f_1 \lor f_2
  \]

• The learning problem:
  
  *Find* \( f_1, f_2, \ldots, f_k \) *that maximizes the performance of program analysis over codebase*

• Our algorithm reduces the search space from \( S^k \) to \( k \cdot S \)
  while formally guaranteeing to preserve global maxima

• Efficient algorithm for searching the subspace \( S \) via iterative and greedy refinement