PREfix

Optional reading: A Static Analyzer for Finding Dynamic Programming Errors

15-413
Introduction to Software Engineering
Jonathan Aldrich

Find the Bugs!

```c
char *f(int size) {
    char * result;
    if (size > 0)
        result = (char *)malloc(size);
    if (size == 1)
        return NULL; // memory leak
    result[0] = 0; // result may be uninitialized
    return result; // malloc may have failed
}
```
Motivation

- Finding programming errors
  - invalid pointers
  - storage allocation errors
  - uninitialized memory
  - improper operations on resources

Can’t we just test?

- 90% of errors involve interactions of multiple functions
  - Is this why the original developer didn’t find them?
- Occur in unusual or error conditions
  - Often hard to exercise with testing
Problems with Other Tools

- False Negatives
  - They look only in one function and miss errors
- False Positives
  - They report errors that can’t really occur
- Hard to use
  - Require extensive program annotations
- Require test cases
  - May be impractical
  - Only as good as your test suite

Goals of PREfix

- Handle hard aspects of C-like languages
  - Pointers, arrays, unions, libraries, casts…
- Don't require user annotations
  - Build on language semantics
- Avoid false positives
  - Use path-sensitive analysis
- Give the user good feedback
  - Why might an error occur? Show the user an example execution
PREfix Analysis

- Explore paths through function
- For each path:
  - Symbolically execute path
  - Determine facts true along the path
  - Compute a guard
  - What must be true for the path to be taken
  - Compute constraints
  - Preconditions for successful execution of path
  - Compute result
  - What is true of the return value?

PREfix: Analysis Example
(syntax slightly de-LISP-ified)

```c
char *f(int size) {
    char * ptr;
    if (size > 0)
        ptr=(char*)malloc(size);
    if (size == 1)
        return NULL;
    ptr[0] = 0;
    return ptr;
}
```

```c
f (param size)
alternate 0
    guard size <= 0
    constraint initialized(size)
    ARRAY ACCESS ERROR: ptr not initialized
alternate 1
    guard size == 1
    constraint initialized(size)
    fact ptr==memory_new(size)
    result return==NULL
    MEMORY LEAK ERROR:
    memory pointed to by ptr is not reachable
    through externally visible state
alternate 2
    guard size > 1
    constraint initialized(size)
    fact ptr==NULL
    ARRAY ACCESS ERROR: ptr is NULL
alternate 3
    guard size > 1
    constraint initialized(size)
    fact ptr==memory_new(size)
    fact ptr[0] == 0
    result return==memory_new(size) && return[0] == 0
alternate 4...
```
Big Ideas

- **Path sensitivity**
  - Avoids reporting errors that occur on control-flow paths that can’t really be taken
- **Dynamic analysis**
  - Explores a *subset* of possible program executions
  - May not find all errors, but still useful
  - Carefully constructed to cover more functionality than most testing strategies can
- **Interprocedural analysis**
  - Looks at how the behavior of a callee affects the caller

Path-Sensitive Analysis

**Analyzes each feasible program path separately**

- **Benefit**
  - Increased precision from eliminating infeasible paths
- **Cost**
  - Exponential number of paths
- **Loops**
  - Infinite number of paths—cannot explore them all
Path Sensitivity: Addressing the Cost

• How does PREfix deal with
  • Exponential path blowup?
    • Explore up to a fixed number of paths
    • Merge paths with identical results
  • Loops
    • Explore up to a fixed number of iterations

What if you miss a path?

```c
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}
```

```c
f (param size)
alternate 0
  guard size <= 0
  constraint initialized(size)
  ARRAY ACCESS ERROR: ptr not initialized
alternate 1
  guard size == 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  result return==NULL
  MEMORY LEAK ERROR:
  memory pointed to by ptr is not reachable
  through externally visible state
alternate 2
  guard size > 1
  constraint initialized(size)
  fact ptr==NULL
  ARRAY ACCESS ERROR: ptr is NULL
alternate 3
  guard size > 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  fact ptr[0] == 0
  result return == memory_new(size) && return[0] == 0
alternate 4...
```
Motivation: Interprocedural Analysis

void exercise_deref() {
    int v = 5;
    int x = deref(&v);
    int y = deref(NULL);
    int z = deref((int *) 5);
}

• Are there errors in this code?
  • Depends on what the function does
  • Second call: error if dereference w/o NULL check
  • Third call: error if any dereference

Interprocedural Analysis

• Any analysis where the analysis results for a caller depend on the results for a callee, or vice versa
Summaries

- Summarize what a function does
  - Maps arguments to results
  - May case-analyze on argument information
  - Simulateable
    - Given information about arguments, will yield:
      - Any errors
      - Information about results

PREfix: Building a Summary

(syntax slightly de-LISP-ified)

```c
int deref(int *p) {
    if (p == NULL)
        return NULL;
    return *p;
}
```

- Return statement
deref (param p)
  alternate return_0
guard p==NULL
constraint initialized(p)
result return==NULL
alternate return_X
guard p != NULL
constraint initialized(p)
constraint valid_ptr(p)
constraint initialized(*p)
result return==*p
void exercise_deref(int v) {
    int v = 5;
    int x = deref(&v);
    int y = deref(NULL);
    int z = deref((int *) 5);
}

deref (param p)
    alternate return_0
        guard p==NULL
            constraint initialized(p)
            result return==NULL
    alternate return_X
        guard p != NULL
            constraint initialized(p)
            constraint valid_ptr(p)
            constraint initialized(*p)
            result return==*p

• Apply summary

exercise_deref
    fact initialized(v), v==5
    fact initialized(&v), valid_ptr(&v)
    fact x==5

• only return_X applies
  • constraint initialized(&v) – PASS
  • constraint valid_ptr(&v) – PASS
  • constraint initialized(*&v) – PASS
  • apply result

int x = deref(&v);

• Apply summary

exercise_deref
    fact initialized(v), v==5
    fact initialized(&v), valid_ptr(&v)
    fact x==5

• only return_0 applies
  • constraint initialized(p) – PASS
  • apply result
PREfix: Using a Summary
(syntax slightly de-LISP-ified)

```c
void exercise_deref(int v) {
    int v = 5;
    int x = deref(&v);
    int y = deref(NULL);
    int z = deref((int *) 5);
}

deref (param p)
    alternate return_0
        guard p==NULL
            constraint initialized(p)
            result return==NULL
    alternate return_X
        guard p != NULL
            constraint initialized(p)
            constraint valid_ptr(p)
            constraint initialized(*p)
            result return=*p
```

- **Apply summary**
  - exercise_deref
    - fact initialized(v), v==5
    - fact initialized(&v), valid_ptr(&v)
    - fact x==5
    - fact y==NULL
    - fact lvalid_ptr((int *) 5), (int *) 5 != NULL
- return_0 does not apply
- return_X does apply
  - constraint initialized((int *) 5) – PASS
  - constraint valid_ptr((int *) 5) – FAIL
    - Generate error

PREfix Scaleability

<table>
<thead>
<tr>
<th>Program</th>
<th>Language</th>
<th>number of files</th>
<th>number of lines</th>
<th>PREfix parse time</th>
<th>PREfix simulation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mozilla</td>
<td>C++</td>
<td>603</td>
<td>540613</td>
<td>2 hours 28 minutes</td>
<td>8 hours 27 minutes</td>
</tr>
<tr>
<td>Apache</td>
<td>C</td>
<td>69</td>
<td>48393</td>
<td>6 minutes</td>
<td>9 minutes</td>
</tr>
<tr>
<td>GDI Demo</td>
<td>C</td>
<td>9</td>
<td>2655</td>
<td>1 second</td>
<td>15 seconds</td>
</tr>
</tbody>
</table>

Table 1: Performance on Sample Public Domain Software

- Analysis cost = 2x-5x build cost
- Scales linearly
  - Probably due to fixed cutoff on number of paths
Value of Interprocedural Analysis

<table>
<thead>
<tr>
<th>model set</th>
<th>execution time (minutes)</th>
<th>statement coverage</th>
<th>branch coverage</th>
<th>predicate coverage</th>
<th>total warning count</th>
<th>using uninit memory</th>
<th>NULL pointer deref</th>
<th>memory leak</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>12</td>
<td>90.1%</td>
<td>87.8%</td>
<td>83.9%</td>
<td>15</td>
<td>2</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>system</td>
<td>13</td>
<td>88.9%</td>
<td>86.3%</td>
<td>82.1%</td>
<td>25</td>
<td>6</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>system &amp; auto</td>
<td>23</td>
<td>73.1%</td>
<td>73.1%</td>
<td>68.6%</td>
<td>248</td>
<td>110</td>
<td>24</td>
<td>124</td>
</tr>
</tbody>
</table>

Table III: Relationships between Available Models, Coverage, Execution Time, and Defects Reported

- 90% of errors require models (summaries)

You don’t need every path

- Get most of the warnings with 100 paths
Empirical Observations

- PREfix finds errors off the main code paths
  - Main-path errors caught by careful coding and testing
- UI is essential
  - Text output is hard to read
  - Need tool to visualize paths, sort defect reports
- Noise warnings
  - Real errors that users don’t care about
    - E.g., memory leaks during catastrophic shutdown

PREfix Summary

- Great tool to find errors
  - Can’t guarantee that it finds them all
    - Role for other tools (e.g., Fluid)
  - Complements testing by analyzing uncommon paths
  - Focuses on low-level errors, not logic/functionality errors
    - Role for functional testing
- Huge impact
  - Used widely within Microsoft
  - Lightweight version is part of new Visual Studio