Isolating Cause–Effect Chains

Andreas Zeller
double bug(double z[], int n) {
    int i, j;

    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
What is the cause of this failure?
1. The programmer creates a defect – an error in the code.

2. When executed, the defect creates an infection – an error in the state.

3. The infection propagates.

4. The infection causes a failure.

This infection chain must be traced back – and broken.
Tracing Infections

- For every infection, we must find the earlier infection that causes it.
- Program analysis tells us possible causes
Tracing Infections
Real Code

- Opaque – e.g. third-party code
- Parallel – threads and processes
- Distributed – across multiple machines
- Dynamic – e.g. reflection in Java
- Multilingual – say, Python + C + SQL
struct foo {
    int tp, len;
union {
    char c[1];
    int i[1];
    struct foo *p[1];
}}
Isolating Input

Difference causes failure
Isolating States

Variables

Difference causes failure
Comparing States

• What is a program state, anyway?
• How can we compare states?
• How can we narrow down differences?
A Sample Program

$ sample 9 8 7
Output: 7 8 9

$ sample 11 14
Output: 0 11

Where is the defect which causes this failure?
int main(int argc, char *argv[]) {
    int *a;

    // Input array
    a = (int *)malloc((argc - 1) * sizeof(int));
    for (int i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    // Sort array
    shell_sort(a, argc);

    // Output array
    printf("Output: ");
    for (int i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
A sample state

• We can access the entire state via the debugger:
  1. List all base variables
  2. Expand all references…
  3. …until a fixpoint is found
Sample States

At the beginning of `shell_sort`, we obtain these states:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>argc</code></td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td><code>argv[0]</code></td>
<td>&quot;./sample&quot;</td>
<td>&quot;./sample&quot;</td>
</tr>
<tr>
<td><code>argv[1]</code></td>
<td>&quot;9&quot;</td>
<td>&quot;11&quot;</td>
</tr>
<tr>
<td><code>argv[2]</code></td>
<td>&quot;8&quot;</td>
<td>&quot;14&quot;</td>
</tr>
<tr>
<td><code>argv[3]</code></td>
<td>&quot;7&quot;</td>
<td>0x0 (NIL)</td>
</tr>
<tr>
<td><code>i</code></td>
<td>1073834752</td>
<td>1073834752</td>
</tr>
<tr>
<td></td>
<td>1074077312</td>
<td>1074077312</td>
</tr>
<tr>
<td><code>j</code></td>
<td>1961</td>
<td>1961</td>
</tr>
<tr>
<td><code>h</code></td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td><code>size</code></td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

This state difference is both effect with the input) as well as cause for the failure).

```c
at shell_sort()
```
Narrowing Down State

Delta Debugging narrows down failure-inducing state changes:

- ■ = \( \delta \) is applied,
- □ = \( \delta \) is not applied

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>2</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
</tr>
<tr>
<td>3</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>4</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>■</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>5</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>6</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>7</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>8</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
</tbody>
</table>

Result: ■
Complex State

• Accessing the state as a *table* is not enough:
  • References are not handled
  • Aliases are not handled
• We need a *richer* representation
A Memory Graph
Structure

Memory Graph

Edge

Operation

Vertex

+value: string
+type: string
+address: void *

0..*
1
2
0..*

root
1
1

Structure

<br><br>

p

0x1234

7
Construction

- Start with `<root>` node and base variables
  - Base variables are on the stack and at fixed locations
- Expand all references, checking for aliases…
- …until all accessible variables are unfolded
Unfolding Memory

• Any variable: make new node
• Structures: unfold all members
• Arrays: unfold all elements
• Pointers: unfold object being pointed to
  • *Does p point to something? And how many?*
Comparing States

passing run

failing run
Comparing States

• Basic idea: compute common subgraph

• Any node that is not part of the common subgraph becomes a difference

• Applying a difference means to create or delete nodes – and adjust references

• All this is done within GDB
Applying Diffs

δ₁₅ creates a variable, δ₂₀ deletes another
Causes in State

Infected state

Sane state

The difference causes GCC to crash!
Search in Space

Infected state

Sane state

Test

Mixed state
Search in Space

Delta Debugging Log

Tests executed

Deltas

cpass
cfail
Search in Space

**Delta Debugging Log**

```
first_loop_store_insn \rightarrow fld[1].rtx \rightarrow fld[1].rtx \rightarrow fld[3].rtx \rightarrow fld[1].rtx \rightarrow code == PLUS
```
Search in Space

Infected state

Sane state

Mixed state

Test

✔

✘
Search in Space

Infected state

Sane state

Mixed state

Test
Search in Time

Failing run

<PLUS node>

Passing run

<PLUS node>
Search in Time

Failing run

Passing run

\[ \text{link} \rightarrow \text{fld}[0].\text{rtx} \rightarrow \text{fld}[0].\text{rtx} == \text{link} \]
Search in Time

Failing run

<PLUS node>

<PLUS node>

<Tree cycle>

Passing run

$t$
Igor has finished debugging your program.

This is what happens in your program when it is invoked as "cc1 -O fail.i".

1. Execution reaches line 4755 of toplevel.c in main.
   Since the program was invoked as "cc1 -O fail.i",
   local variable argv[2] is now "fail.i".

2. Execution reaches line 470 of combine.c in combine_instructions.
   Since argv[2] was "fail.i",
   variable first_loop_store_insn->fld[1].rtx->fld[1].rtx->
   fld[3].rtx->fld[1].rtx now points to a new rtx_def.

3. Execution reaches line 6761 of combine.c in if_then_else_cond.
   Since first_loop_store_insn->fld[1].rtx->fld[1].rtx->
   fld[3].rtx->fld[1].rtx pointed to a new rtx_def,
   variable link->fld[0].rtx->fld[0].rtx is now link.

4. Execution ends.
   Since link->fld[0].rtx->fld[0].rtx was link,
   the program crashes with a SIGSEGV signal.
   The program fails.

Need more details? Select the effects you want to focus upon and
Plain wrong? Please check the failure symptoms as determined by Igor.
Any questions? See the AskIgor Forum!
Capturing State
for Python programs

if __name__ == "__main__":
    sys.settrace(tracer)
    ...

def tracer(frame, event, arg):
    dump_stack(frame)
    return tracer
def dump_stack(frame):
    while frame is not None:
        dump_frame(frame)
        frame = frame.f_back

def dump_frame(frame):
    locals = frame.f_locals
    globals = frame.f_globals
    print locals, globals
def dump_frame(frame):
    locals = frame.f_locals
    locals['a'] = 42

equivalent to assignment
“a = 42” in frame
Caveats

Python frame objects are translated back to internal frames only after tracer() has returned:

- Frames can be inspected at any time, but changed only in tracer()
- Output of variables during tracer() may inhibit their translation at return
Open Issues

- How do we capture an accurate state?
- How do we ensure the cause is valid?
- Where does a state end?
- What is the cost?
- When do we compare states? (next lecture)
Concepts

★ Delta Debugging on program states isolates a cause-effect chain through the run
★ Use memory graphs to extract and compare program states
★ Demanding, yet effective technique