Isolating and Locating Cause–Effect Chains

Andreas Zeller

**bug.c**

define 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 do we do now?

What is the cause of this failure?
From Defect to 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.
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

And even if we know everything, there still is code which is almost impossible to analyze. In C, for instance, only the programmer knows how memory is structured; there is no general way for static analysis to find this out.

In the last lecture, we have seen delta debugging on input.
Now let’s take a deeper view. If a program is a succession of states, can’t we treat each state as an input to the remainder of the run?

Let’s look at a simpler example first.
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

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>a[0]</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>a[1]</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>a[2]</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>a'[0]</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>a'[1]</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>a'[2]</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>a'[3]</td>
<td>1961</td>
<td>1961</td>
</tr>
</tbody>
</table>

at shell_sort()
Narrowing State Diffs

= δ is applied, □ = δ is not applied

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>7</td>
<td>✔</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td>☒</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>11</td>
<td>☒</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>11</td>
<td>☐</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>14</td>
<td>☒</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>0</td>
<td>☒</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>8</td>
<td>☒</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>9</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
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

- 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

\( \delta_{15} \) creates a variable, \( \delta_{20} \) deletes another

State of the GNU compiler (GCC)
42991 vertices
44290 edges - and 1 is wrong :-)
An actual GCC execution has millions of these states.

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

Transitions

A *cause transition* occurs when a *new variable* begins to be a failure cause:

- **PLUS** no longer causes the failure…
- …but **the tree cycle** does!

Can be narrowed down by binary search

Why Transitions?

- Each failure cause in the program state is caused by some statement
- These statements are executed at *cause transitions*
- Cause transitions thus are statements that cause the failure!

Potential Fixes

- Each cause transition implies a *fix* to make the failure no longer occur – just prohibit the transition
- A cause transition is more than a potential fix – it may be “the” defect itself
All GCC Transitions

<table>
<thead>
<tr>
<th>Location</th>
<th>Cause transition to variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Start)</td>
<td>argy[3]</td>
</tr>
<tr>
<td>1 toplev.c:4755</td>
<td>name</td>
</tr>
<tr>
<td>2 toplev.c:2909</td>
<td>dump_base_name</td>
</tr>
<tr>
<td>3 c-lex.c:187</td>
<td>finput-&gt;i0_buf_base</td>
</tr>
<tr>
<td>4 c-lex.c:1213</td>
<td>nextchar</td>
</tr>
<tr>
<td>5 c-lex.c:1213</td>
<td>yysa[s1]</td>
</tr>
<tr>
<td>6 c-typeck.c:3615</td>
<td>yysa[42]</td>
</tr>
<tr>
<td>7 c-lex.c:1213</td>
<td>last_insn-&gt;fld[1].rtx</td>
</tr>
<tr>
<td>8 c-lex.c:1213</td>
<td>sequence_result[2]</td>
</tr>
<tr>
<td>9 combine.c:4271</td>
<td>x=fld[0].rtx-&gt;fld[0].rtx</td>
</tr>
</tbody>
</table>

combine.c:4279

if (GET_CODE (XEXP (x, 0)) == PLUS {
  x = apply_distributive_law
  gen_binary (PLUS, mode,
    gen_binary (MULT, mode,
      XEXP (XEXP (x, 0), 0),
      XEXP (x, 1)),
    gen_binary (MULT, mode,
      XEXP (XEXP (x, 0), 1),
      XEXP (x, 1)));
  if (GET_CODE (x) != MULT) Should be copy_rtx()
    return x;
}
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?

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

Concepts

★ Cause transitions pinpoint failure causes in the program code
★ Failure-causing statements are potential fixes (and frequently defects, too)
★ Even more demanding, yet effective technique