Finding Causes

Infected state  Sane state

The difference causes the failure

Search in Space

Infected state  Sane state

Test

Mixed state

 argc = 3

 ✔

 ✘

 Test?
Search in Time

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

- `argc` no longer causes the failure...
- ...but `a[2]` does!

Can be narrowed down by binary search

```c
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;
}
```
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.

Searching GCC State

[Diagram showing infected, sane, and mixed states with test and PLUS node connections.]
All GCC Transitions

<table>
<thead>
<tr>
<th>#</th>
<th>Location</th>
<th>Cause transition to variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(Main)</td>
<td><code>aryv[2]</code></td>
</tr>
<tr>
<td>1</td>
<td><code>toplevel.c:4755</code></td>
<td><code>name</code></td>
</tr>
<tr>
<td>2</td>
<td><code>toplevel.c:2909</code></td>
<td><code>dump_base_name</code></td>
</tr>
<tr>
<td>3</td>
<td><code>c-lex.c:187</code></td>
<td><code>finpt-&gt;_20_buf_base</code></td>
</tr>
<tr>
<td>4</td>
<td><code>c-lex.c:1213</code></td>
<td><code>nextchar</code></td>
</tr>
<tr>
<td>5</td>
<td><code>c-lex.c:1213</code></td>
<td><code>yyssa[41]</code></td>
</tr>
<tr>
<td>6</td>
<td><code>c-typeck.c:3615</code></td>
<td><code>yyssa[42]</code></td>
</tr>
<tr>
<td>7</td>
<td><code>c-lex.c:1213</code></td>
<td><code>last_insn-&gt;fld[1].rtx</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>→fld[1].rtx→fld[3].rtx</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>→fld[1].rtx.code</code></td>
</tr>
<tr>
<td>8</td>
<td><code>c-decl.c:1213</code></td>
<td><code>sequence_result[2]</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>→fld[0].rtvec</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>→elem[0].rtx→fld[1].rtx</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>→fld[1].rtx→fld[1].rtx</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>→fld[1].rtx→fld[1].rtx</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>→fld[1].rtx→fld[1].rtx</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>→fld[3].rtx→fld[1].rtx</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>→fld[3].rtx→fld[1].rtx.code</code></td>
</tr>
<tr>
<td>9</td>
<td><code>combine.c:4271</code></td>
<td><code>x→fld[0].rtx→fld[0].rtx</code></td>
</tr>
</tbody>
</table>

combine.c:4279

```c
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;
}
```

How good are we?

Evaluation using the Siemens Testsuite:

- 7 programs – most text processors
- 132 variations, each with 1 seeded defect
- Challenge: Using test runs, locate defect
- All proposed defect locators fail
  (Comparing coverage, slicing, dynamic invariants)
Close to the Defect

Predicted location

Locating Defects

- NN (Nearest Neighbor) @Brown by Manos Renieris + Stephen Reiss
- CT (Cause Transitions) @Saarland by Holger Cleve + Andreas Zeller
- SD (Statistical Debugging) @Berkeley by Ben Liblit (now Wisconsin), Mayur Naik (Stanford), Alice Zheng, Alex Aiken (now Stanford), Michael Jordan
- SOBER @Urbana-Champaign + Purdue by

Open Issues

- Hierarchical search
- Ranking transitions
- User-side diagnosis
- Combination with statistical causality
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