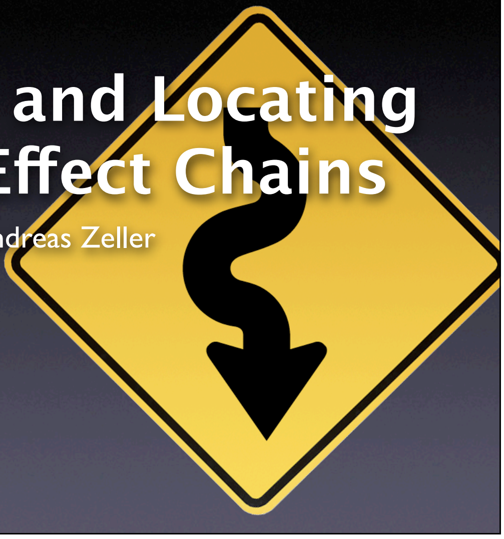
Isolating and Locating Cause-Effect Chains

Andreas Zeller



1

bug.c

```
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];
}
```


2

braeburn: bug — bash — 80×24

$

What is the cause of this failure?

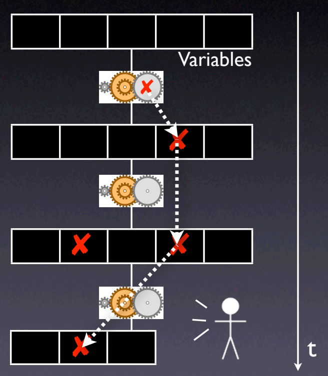
3

What do we do now?

From Defect to Failure

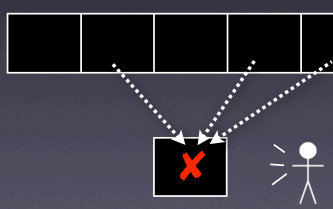
1. The programmer creates a *defect* – an error in the code.
2. When executed, the defect creates an *infection* – an error in the state.
3. The infection *propagates*.
4. The infection causes a *failure*.

This infection chain must be traced back – and broken.

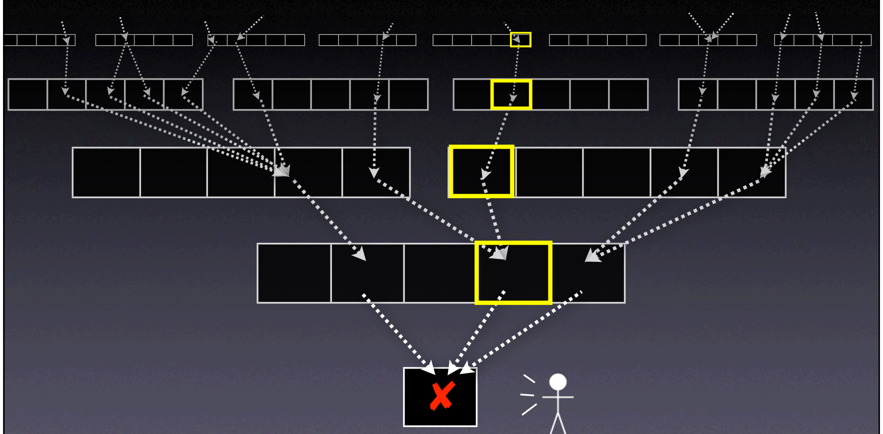


Tracing Infections

- For every infection, we must find the *earlier infection* that *causes* it.
- Program analysis tells us *possible causes*



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

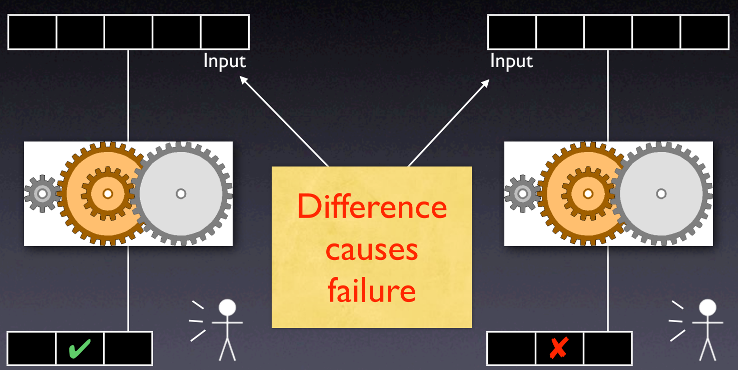
Obscure Code

```
struct foo {
    int tp, len;
    union {
        char       c[1];
        int        i[1];
        struct foo *p[1];
}}
```

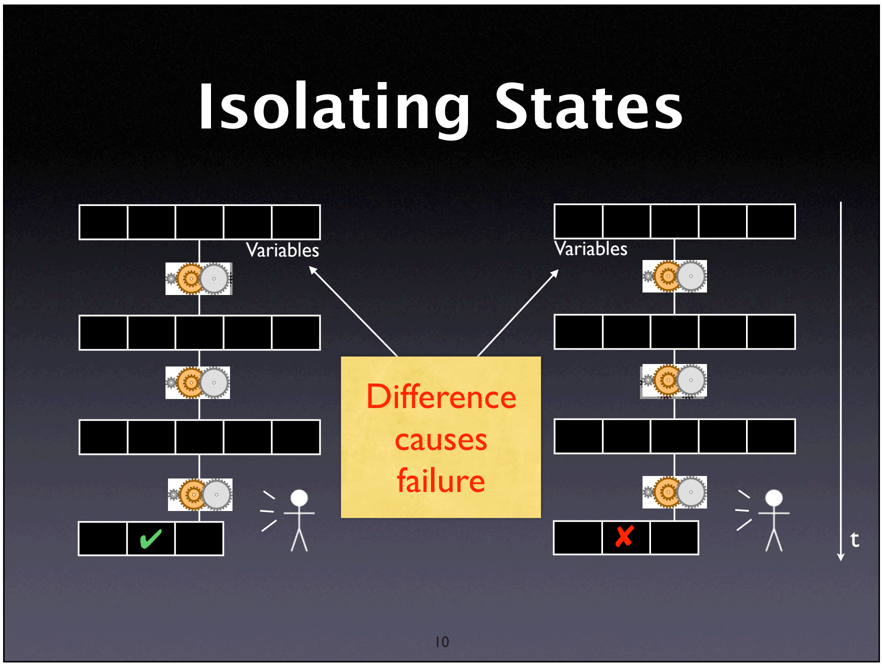
tp	len	p[0]

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

Isolating Input

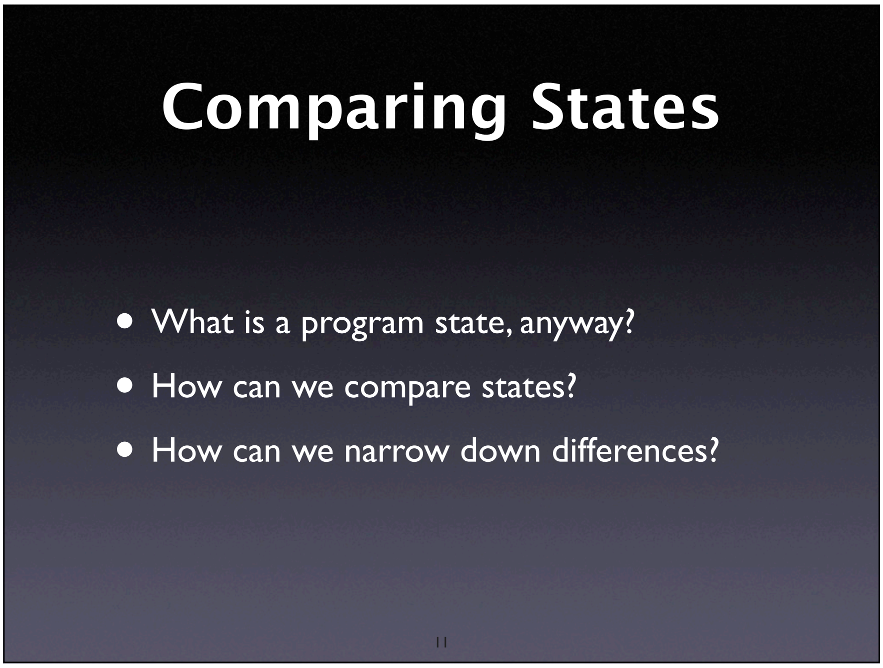


In the last lecture, we have seen delta debugging on input.

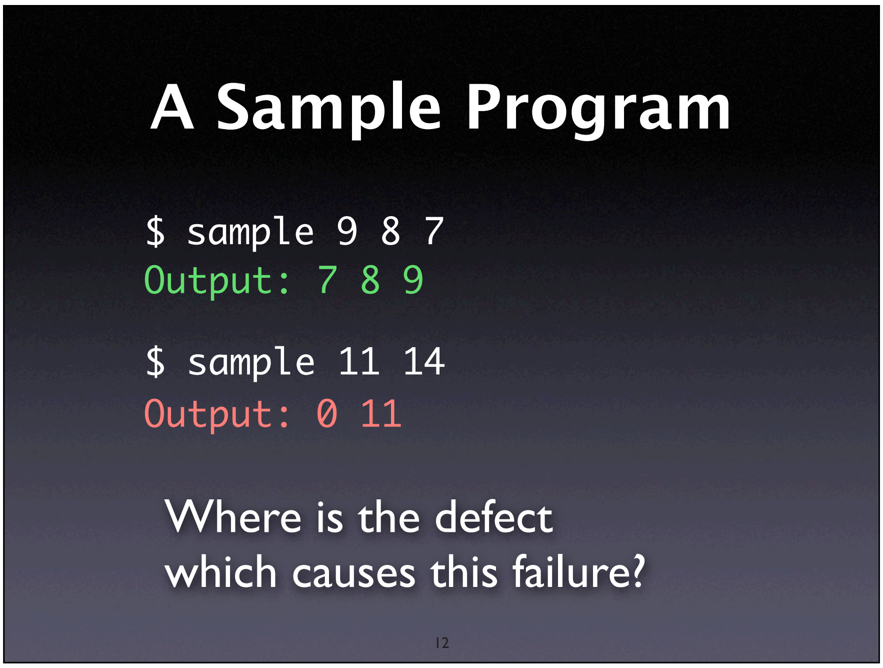


Now let’s take a deeper view. If a program is a succession of states, can’t we treat each state as **an input to the remainder of the run?**

10



11



Let’s look at a simpler example first.

12

```
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

Variable	Value in $r_{\checkmark}$	Value in $r_{\times}$
argc	4	5
argv[0]	"./sample"	"./sample"
argv[1]	"9"	"11"
argv[2]	"8"	"14"
argv[3]	"7"	0x0 (NIL)
i′	1073834752	1073834752
j	1074077312	1074077312
h	1961	1961
size	4	3

Variable	Value in $r_{\checkmark}$	Value in $r_{\times}$
i	3	2
a[0]	9	11
a[1]	8	14
a[2]	7	0
a[3]	1961	1961
a′[0]	9	11
a′[1]	8	14
a′[2]	7	0
a′[3]	1961	1961

at shell_sort()

Narrowing State Diffs

■ = δ is applied, □ = δ is *not* applied

#	$a'[0]$	$a[0]$	$a'[1]$	$a[1]$	$a'[2]$	$a[2]$	$argc$	$argv[1]$	$argv[2]$	$argv[3]$	i	$size$	Output	Test
1	□	□	□	□	□	□	□	□	□	□	□	□	7 8 9	✔
2	■	■	■	■	■	■	■	■	■	■	■	■	0 11	✘
3	■	■	■	■	■	■	□	□	□	□	□	□	0 11 14	✘
4	■	■	■	□	□	□	□	□	□	□	□	□	7 11 14	?
5	□	□	□	■	■	■	□	□	□	□	□	□	0 9 14	✘
6	□	□	□	■	□	□	□	□	□	□	□	□	7 9 14	?
7	□	□	□	□	■	■	□	□	□	□	□	□	0 8 9	✘
8	□	□	□	□	■	□	□	□	□	□	□	□	0 8 9	✘
Result					■									

16

16

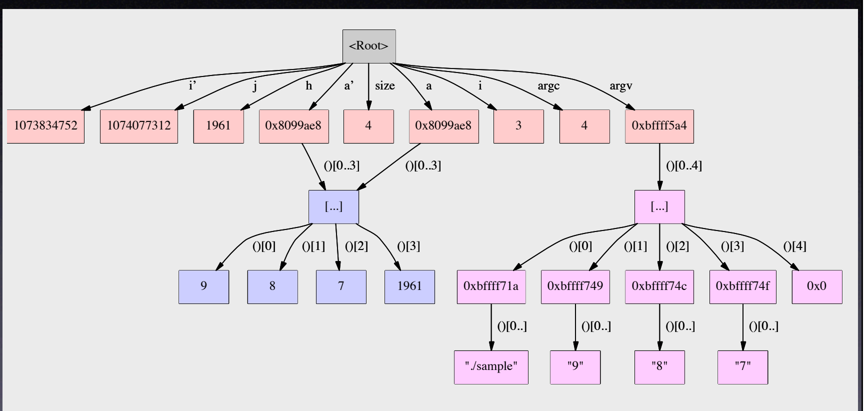
Complex State

- Accessing the state as a *table* is not enough:
  - References are not handled
  - Aliases are not handled
- We need a *richer* representation

17

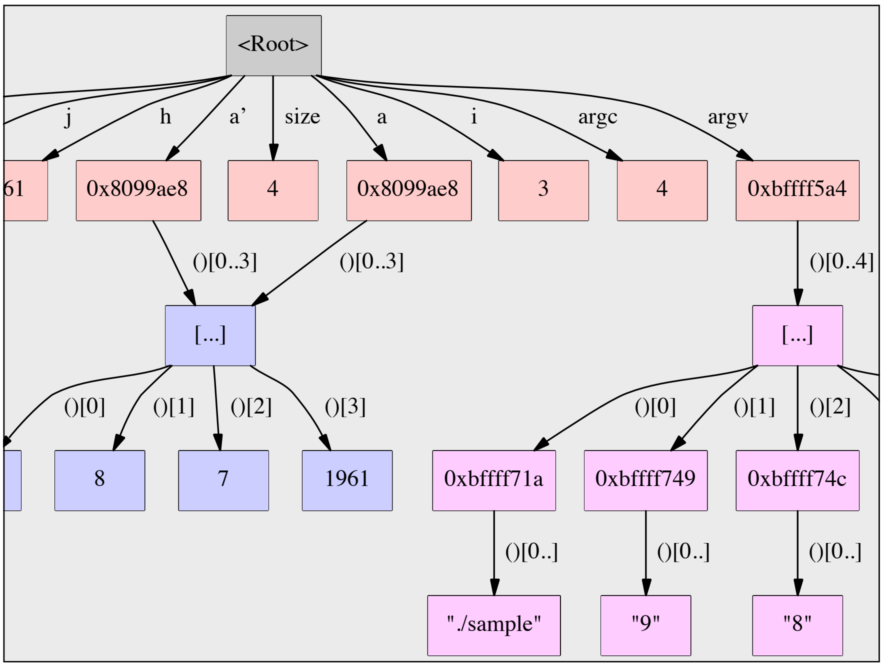
17

A Memory Graph



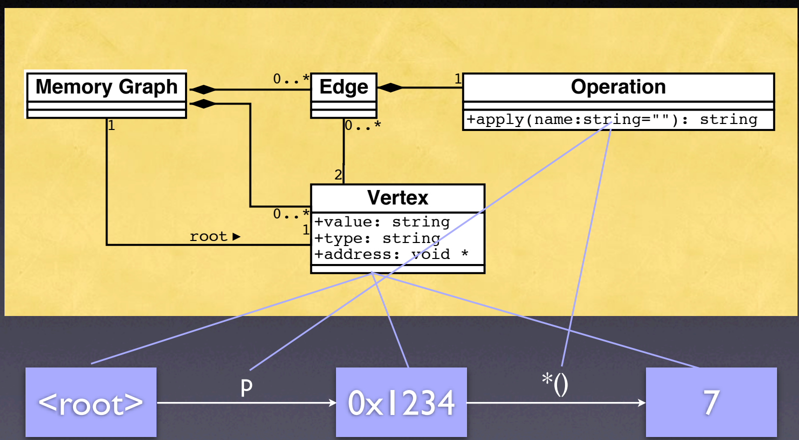
18

18



19

Structure



20

20

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

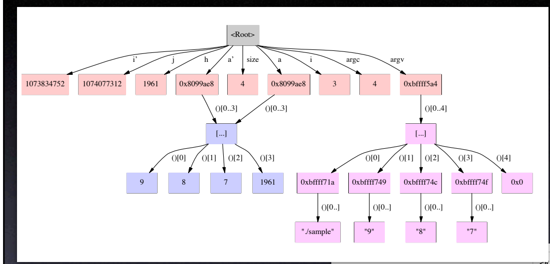
21

21

Unfolding Memory

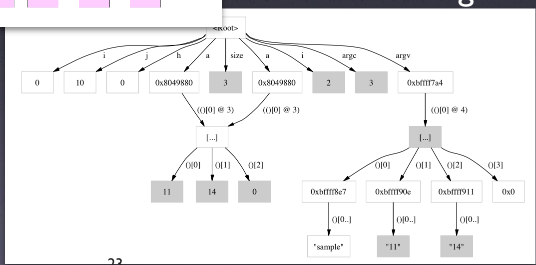
- Any variable: make new node
- Structures: unfold all members
- Arrays: unfold all elements
- Pointers: unfold object being pointed to
  - *Does p point to something? And how many?*

Comparing States



passing run

failing run

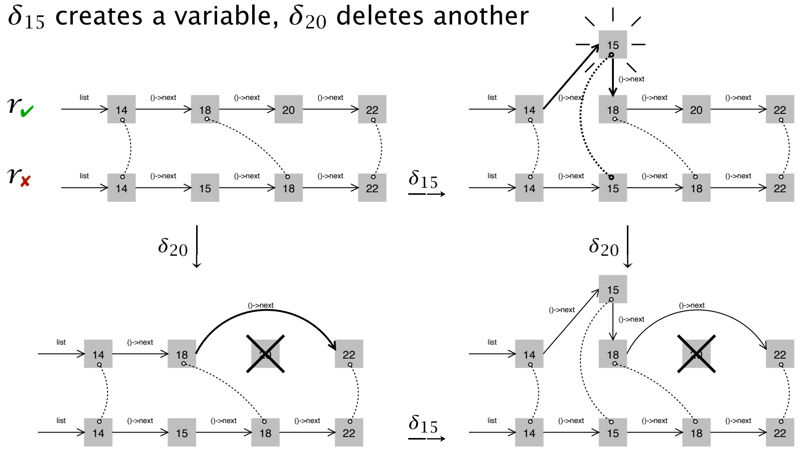


Comparing States

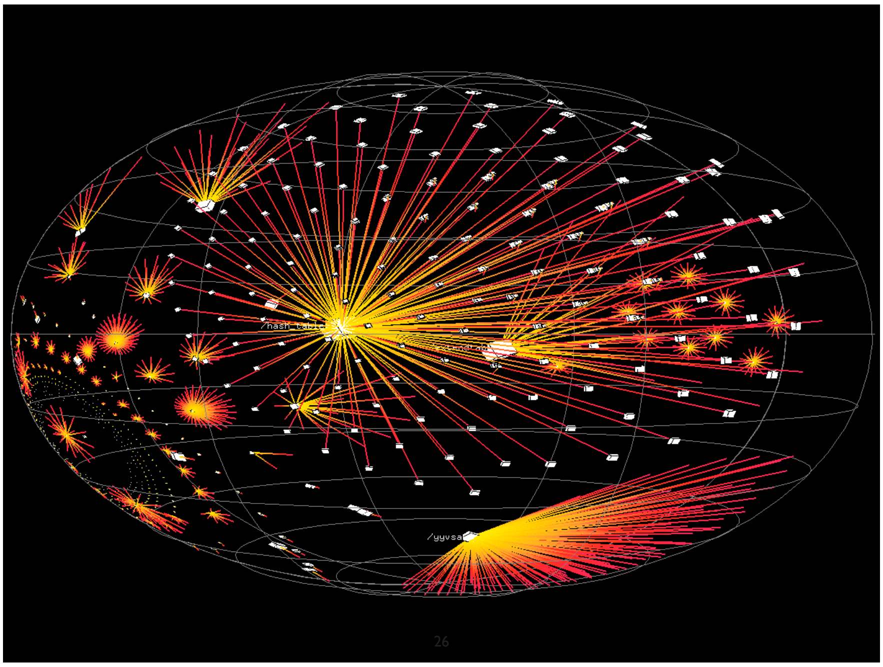
- Basic idea: *compute common subgraph*
- Any node that is not part of the common subgraph becomes a *difference*
- Applying a difference means to create or delete nodes – and adjust references
- All this is done within GDB

Applying Diffs

δ_{15} creates a variable, δ_{20} deletes another



25



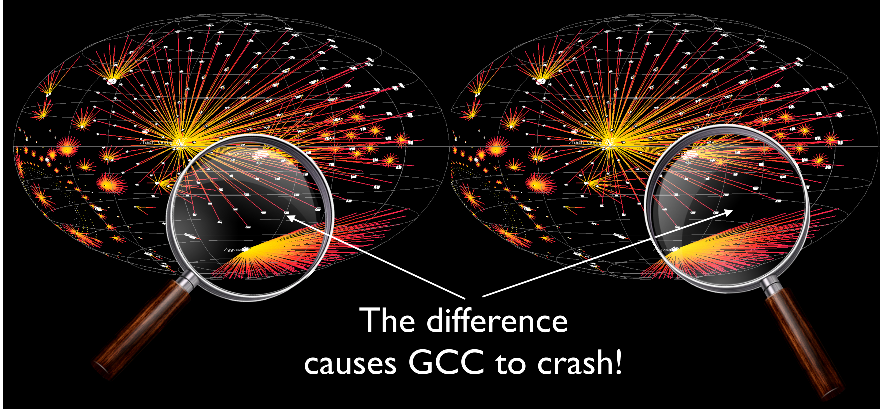
26

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

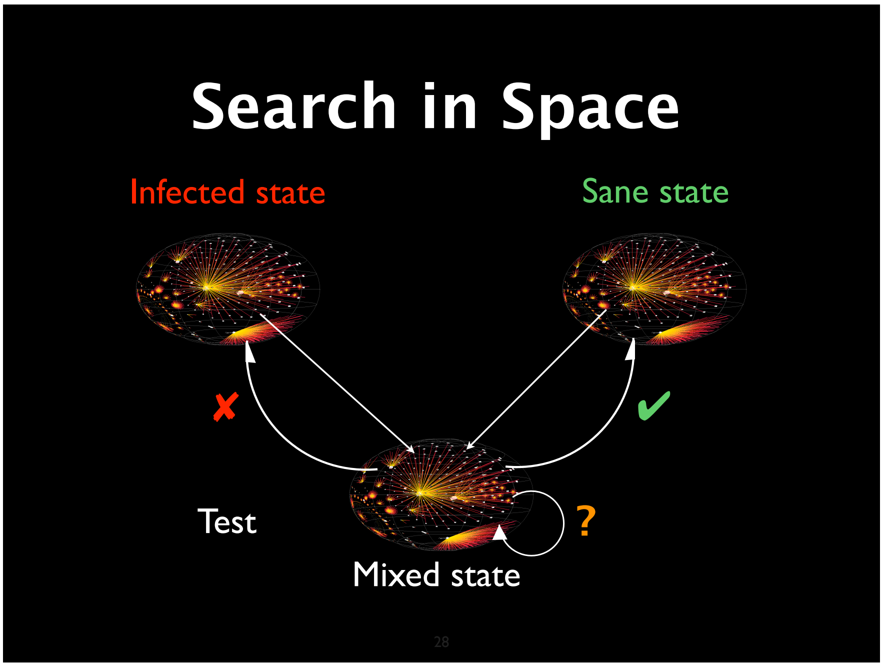
Causes in State

Infected state

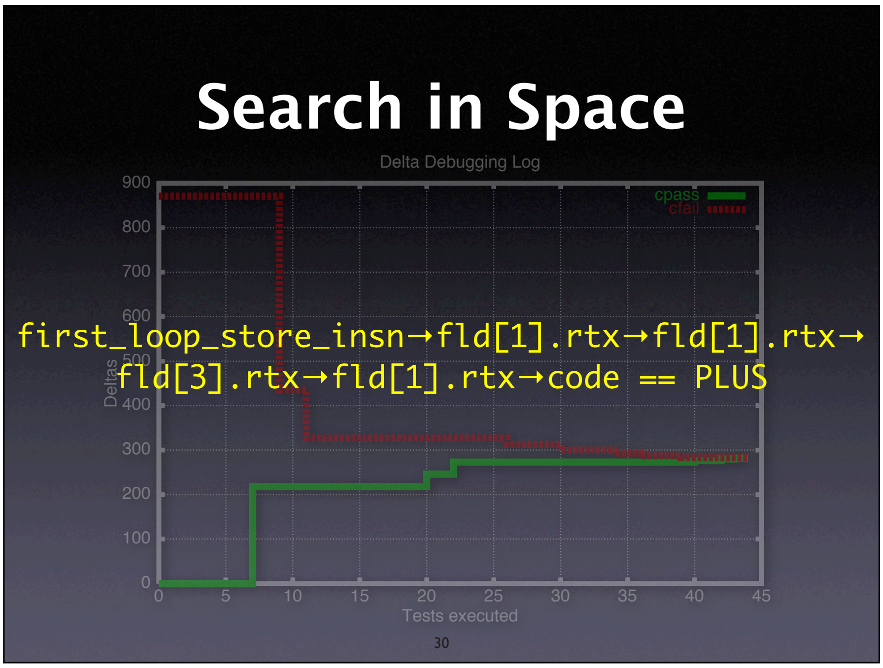
Sane state

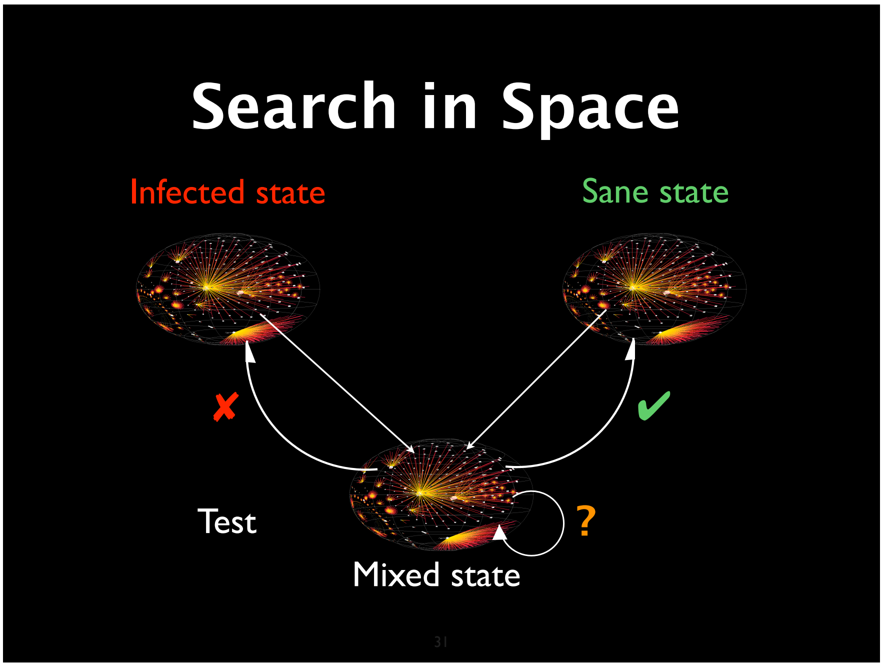


27

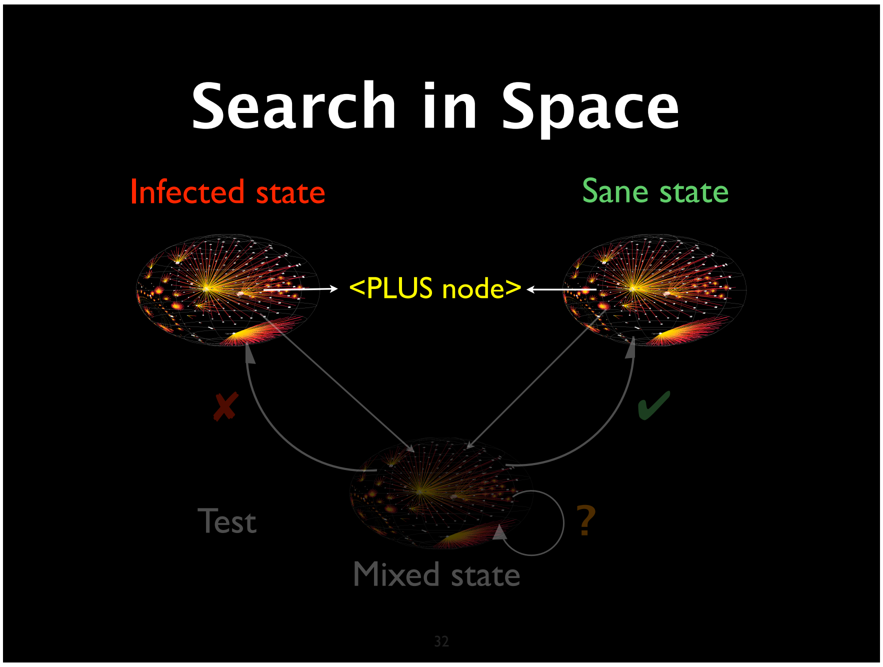

Search in Space
Infected state
Sane state
✘
✔
Test
?
Mixed state


Search in Space
Delta Debugging Log
cpass
cfail
Deltas
Tests executed
900
800
700
600
500
400
300
200
100
0
0
5
10
15
20
25
30
35
40
45

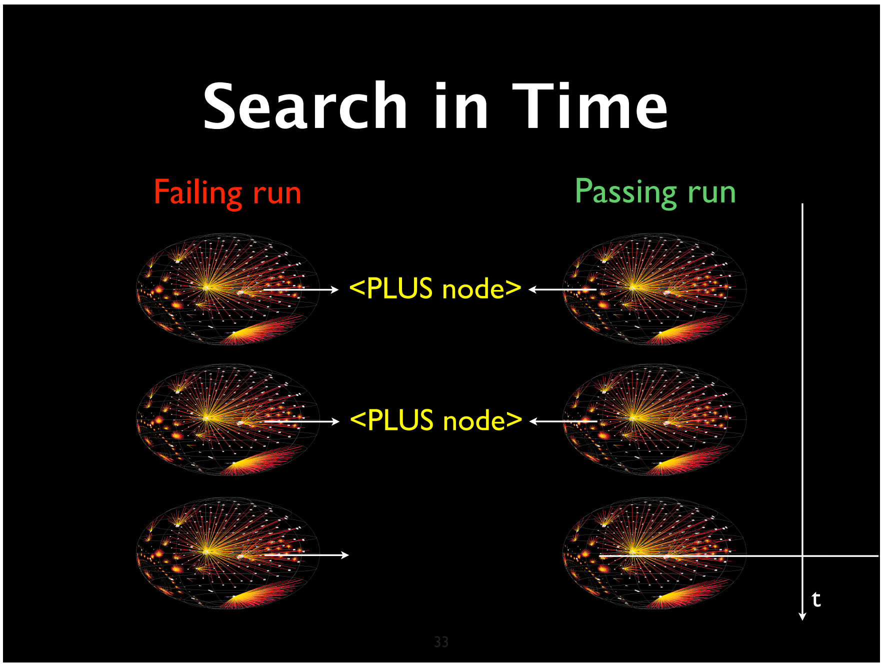

Search in Space
Delta Debugging Log
first_loop_store_insn→fld[1].rtx→fld[1].rtx→
fld[3].rtx→fld[1].rtx→code == PLUS
Tests executed


Search in Space
Infected state
Sane state
✘
✔
Test
?
Mixed state
31

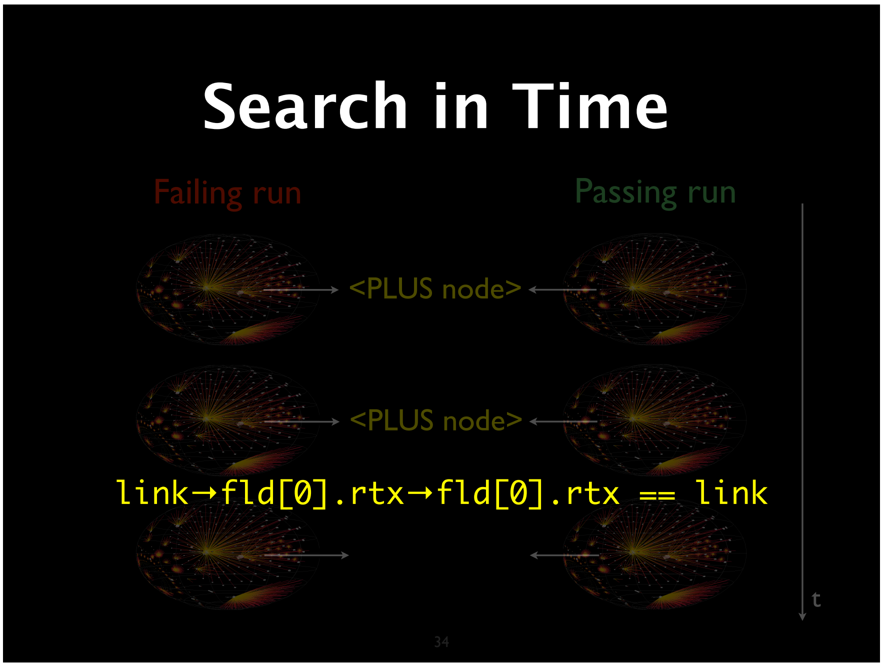
31


Search in Space
Infected state
Sane state
<PLUS node>
✘
✔
Test
?
Mixed state
32

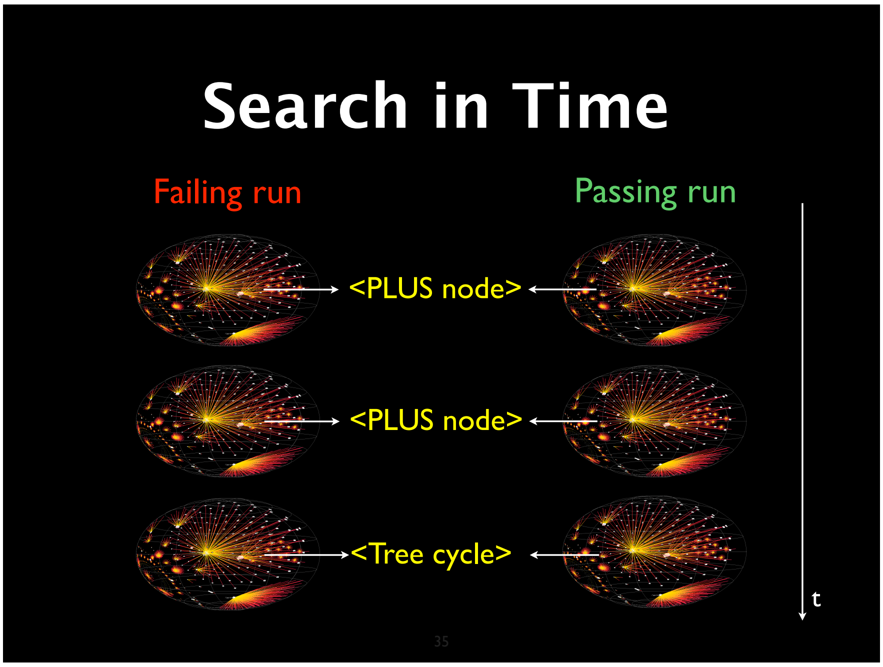
32


Search in Time
Failing run
Passing run
<PLUS node>
<PLUS node>
t
33

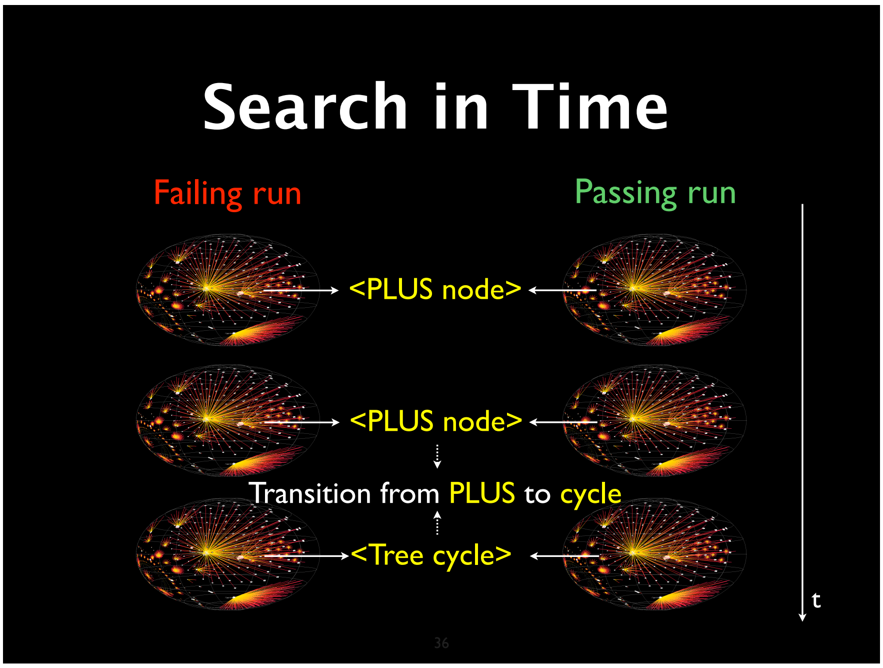
33


Search in Time
Failing run
Passing run
<PLUS node>
<PLUS node>
link→fld[0].rtx→fld[0].rtx == link
t
34

34


Search in Time
Failing run
Passing run
<PLUS node>
<PLUS node>
<Tree cycle>
t
35

35


Search in Time
Failing run
Passing run
<PLUS node>
<PLUS node>
Transition from PLUS to cycle
<Tree cycle>
t
36

36

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

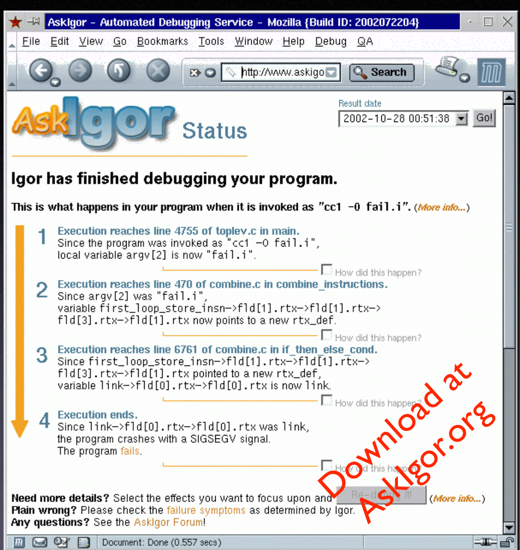
#	Location	Cause transition to variable
0	⟨Start⟩	`argv[3]`
1	toplev.c:4755	`name`
2	toplev.c:2909	`dump_base_name`
3	c-lex.c:187	`finput→_IO_buf_base`
4	c-lex.c:1213	`nextchar`
5	c-lex.c:1213	`yyssa[41]`
6	c-typeck.c:3615	`yyssa[42]`
7	c-lex.c:1213	`last_insn→fld[1].rtx →fld[1].rtx→fld[3].rtx →fld[1].rtx.code`
8	c-decl.c:1213	`sequence_result[2] →fld[0].rtvec →elem[0].rtx→fld[1].rtx →fld[1].rtx→fld[1].rtx →fld[1].rtx→fld[1].rtx →fld[1].rtx→fld[1].rtx →fld[3].rtx→fld[1].rtx.code`
9	combine.c:4271	`x→fld[0].rtx→fld[0].rtx`

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)
         return x;
}
```

Should be copy_rtx()



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