How Failures Come to be

Andreas Zeller
An F–16
(northern hemisphere)
An F–16
(southern hemisphere)
F–16 Landing Gear
The First Bug
September 9, 1947

Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

1545
More Bugs
Facts on Debugging

• Software bugs are costing ~60 bln US$/yr
• Improvements could reduce cost by 30%
• Validation (including debugging) can easily take up to 50-75% of the development time
• When debugging, some people are three times as efficient than others
A Sample Program

$ sample 9 8 7
Output: 7 8 9

$ sample 11 14
Output: 0 11
How to Debug
(Sommerville 2004)

Locate error → Design error repair → Repair error → Re-test program
The Traffic Principle

- **T**rack the problem
- **R**eproduce
- **A**utomate
- **F**ind Origins
- **F**ocus
- **I**solate
- **C**orrect
The Traffic Principle

- Track the problem
- Reproduce
- Automate
- Find Origins
- Focus
- Isolate
- Correct
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.
The Curse of Testing

- Not every defect causes a failure!
- *Testing can only show the presence of errors – not their absence.*
  (Dijkstra 1972)
• Every failure can be traced back to some infection, and every infection is caused by some defect.

• Debugging means to relate a given failure to the defect – and to remove the defect.
Search in Space + Time
The Defect

variables

![Diagram showing variables over time]

- time

The diagram illustrates the concept of variables over time, with a grid representing the passage of time and nodes indicating variable states.
A Program State
A Sample Program

$ sample 9 8 7
Output: 7 8 9

$ sample 11 14
Output: 0 11
int main(int argc, char *argv[])
{
    int *a;
    int i;

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

    shell_sort(a, argc);

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

    free(a);

    return 0;
}
Find Origins

- The 0 printed is the value of a[0]. Where does it come from?
- Basic idea: Track or deduce value origins
- Separates relevant from irrelevant values
- We can trace back a[0] to shell_sort
static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}
Search in Time

- In `shell_sort`, the state must have become infected.
- Basic idea: Observe a transition from sane to infected.
Observing a Run

variables

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>&quot;11&quot;</td>
<td>&quot;14&quot;</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a = malloc(...)
i = 0
a[i] = atoi(argv[i + 1])
i++
a[i] = atoi(argv[i + 1])
i++
shell_sort(a, argc)
return 0
Specific Observation

static void shell_sort(int a[], int size)
{
    fprintf(stderr, “At shell_sort”);
    for (i = 0; i < size; i++)
        fprintf(stderr, “a[%d] = %d\n”, i, a[i]);
    fprintf(stderr, “size = %d\n”, size);
    int i, j;
    int h = 1;
    ...
}

The state is infected at the call of shell_sort!
Fixing the Program

```c
int main(int argc, char *argv[]) {
    int *a;
    int i;

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

    shell_sort(a, argc);  
    $ sample 11 14
    Output: 11 14
    
    ...
}
```
Finding Causes

Infected state  Sane state

The difference causes the failure
Search in Space

Infected state

Sane state

argc = 3

Test

Mixed state
Search in Time

Failing run

Passing run

\[
\begin{align*}
\text{argc} &= 3 \\
\text{argc} &= 3 \\
\text{Transition from argc to a[2]} \\
a[2] &= 0
\end{align*}
\]
```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 - 1);

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

    free(a);
    return 0;
}
```
Igor has finished debugging your program.

This is what happens in your program when it is invoked as "sample 11 14".

1. Execution reaches line 35 of sample.c in main.
   Since argc was 3,
   local variable a[2] is now 0.

2. Execution reaches line 18 of sample.c in shell_sort for the 2nd time.
   Since a[2] was 0,
   local variable v is now 0.

3. Execution reaches line 15 of sample.c in shell_sort for the 2nd time.
   Since v was 0,
   local variable a[0] is now 0.

4. Execution ends.
   Since a[0] was 0,
   the output now contains "0".
   The program fails.

Need more details? Select the effects you want to focus upon and (Re-debug it!) (More info...)
Plain wrong? Please check the failure symptoms as determined by Igor.
Any questions? See the AskIgor Forum!
A failure comes to be in three stages:

1. The programmer creates a *defect*
2. The defect causes an *infection*
3. The infection causes a *failure* -- an externally visible error.

Not every defect results in an infection, and not every infection results in a failure.
To debug a program, proceed in 7 steps:

- **T**rack the problem
- **R**eproduce
- **A**utomate
- **F**ind Origins
- **F**ocus
- **I**solate
- **C**orrect
Concepts (3)

★ A variety of tools and techniques is available to *automate debugging*:

- Program Slicing
- Observing & Watching State
- Asserting Invariants
- Detecting Anomalies
- Isolating Cause-Effect Chains