Simplifying and Isolating Failure-Inducing Input

Andreas Zeller

Saarland University, Germany

Ralf Hildebrandt

DeTeLine – Deutsche Telekom, Germany

Presented by Ting Su

(slides are adapted from Nir Peer, University of Maryland)

Overview

- Yesterday, my program worked. Today, it does not.
- What is the minimal test case that reproduces the failure?
- Also, what is the difference between a passing and a failing test case?

How do we go from

```
<td align=left valign=top>
<SELECT NAME="op sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows 3.1<OPTION VALUE="Windows 95">Windows
95<OPTION VALUE="Windows 98">Windows 98<OPTION VALUE="Windows ME">Windows ME<OPTION VALUE="Windows
2000">Windows 2000<OPTION VALUE="Windows NT">Windows NT<OPTION VALUE="Mac System 7">Mac System
7<OPTION VALUE="Mac System 7.5">Mac System 7.5<OPTION VALUE="Mac System 7.6.1">Mac System
7.6.1<OPTION VALUE="Mac System 8.0">Mac System 8.0<OPTION VALUE="Mac System 8.5">Mac System
8.5<OPTION VALUE="Mac System 8.6">Mac System 8.6<OPTION VALUE="Mac System 9.x">Mac System 9.x<OPTION
VALUE="MacOS X">MacOS X<OPTION VALUE="Linux">Linux<OPTION VALUE="BSDI">BSDI<OPTION
VALUE="FreeBSD">FreeBSD<OPTION VALUE="NetBSD">NetBSD<OPTION VALUE="OpenBSD">OpenBSD<OPTION
VALUE="AIX">AIX<OPTION VALUE="BeOS">BeOS<OPTION VALUE="HP-UX">HP-UX<OPTION VALUE="IRIX">IRIX<OPTION
VALUE="Neutrino">Neutrino<OPTION VALUE="OpenVMS">OpenVMS<OPTION VALUE="OS/2">OS/2<OPTION VALUE="OSF/
1">OSF/1<OPTION VALUE="Solaris">Solaris<OPTION VALUE="SunOS">SunOS<OPTION VALUE="other">other</
SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="--">--<OPTION VALUE="P1">P1<OPTION VALUE="P2">P2<OPTION VALUE="P3">P3<OPTION
VALUE="P4">P4<OPTION VALUE="P5">P5</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="bug severity" MULTIPLE SIZE=7>
<OPTION VALUE="blocker">blocker<OPTION VALUE="critical">critical<OPTION VALUE="major">major<OPTION
VALUE="normal">normal<OPTION VALUE="minor">minor<OPTION VALUE="trivial">trivial<OPTION
VALUE="enhancement">enhancement</SELECT>
</tr>
</table>
```

File → Print → Segmentation Fault

into this

<SELECT>

File → Print → Segmentation Fault

What developers face?

- Mozilla receives several dozens bug reports one day.
- Bugzilla listed more than 370 open bug reports for Mozilla (July 1999).
- Who can help simplify these bug reports?

What developers want?

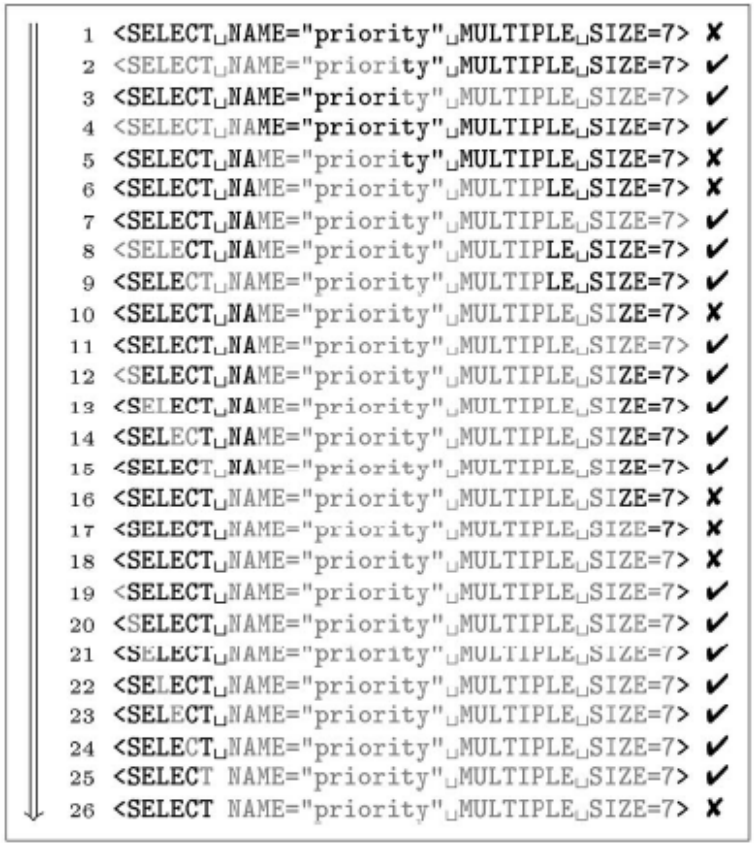
- Turning bug reports into minimal tests.
- The simplest HTML page that reproduces the fault.
- Let‘s automate this task!

Simplification of Test Cases

- Takes a failing test case
- Simplifies it by successive testing
- Stops when a minimal test case is reached
  - remove any single input entity will cause the failure to disappear

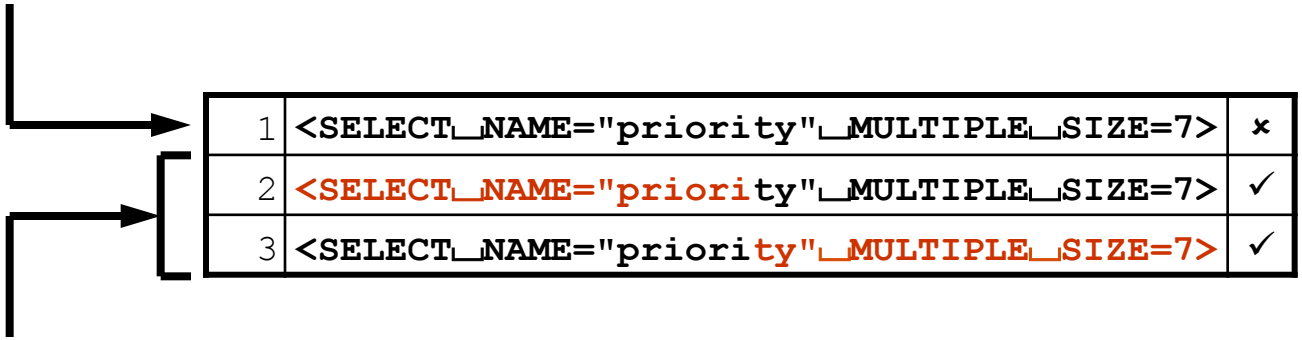
How to minimize a test case?

- Test subsets with removed characters (shown in grey)
- A given test case
  - Fails (✘) if Mozilla crashes on it
  - Passes (✓) otherwise



How to minimize a test case?

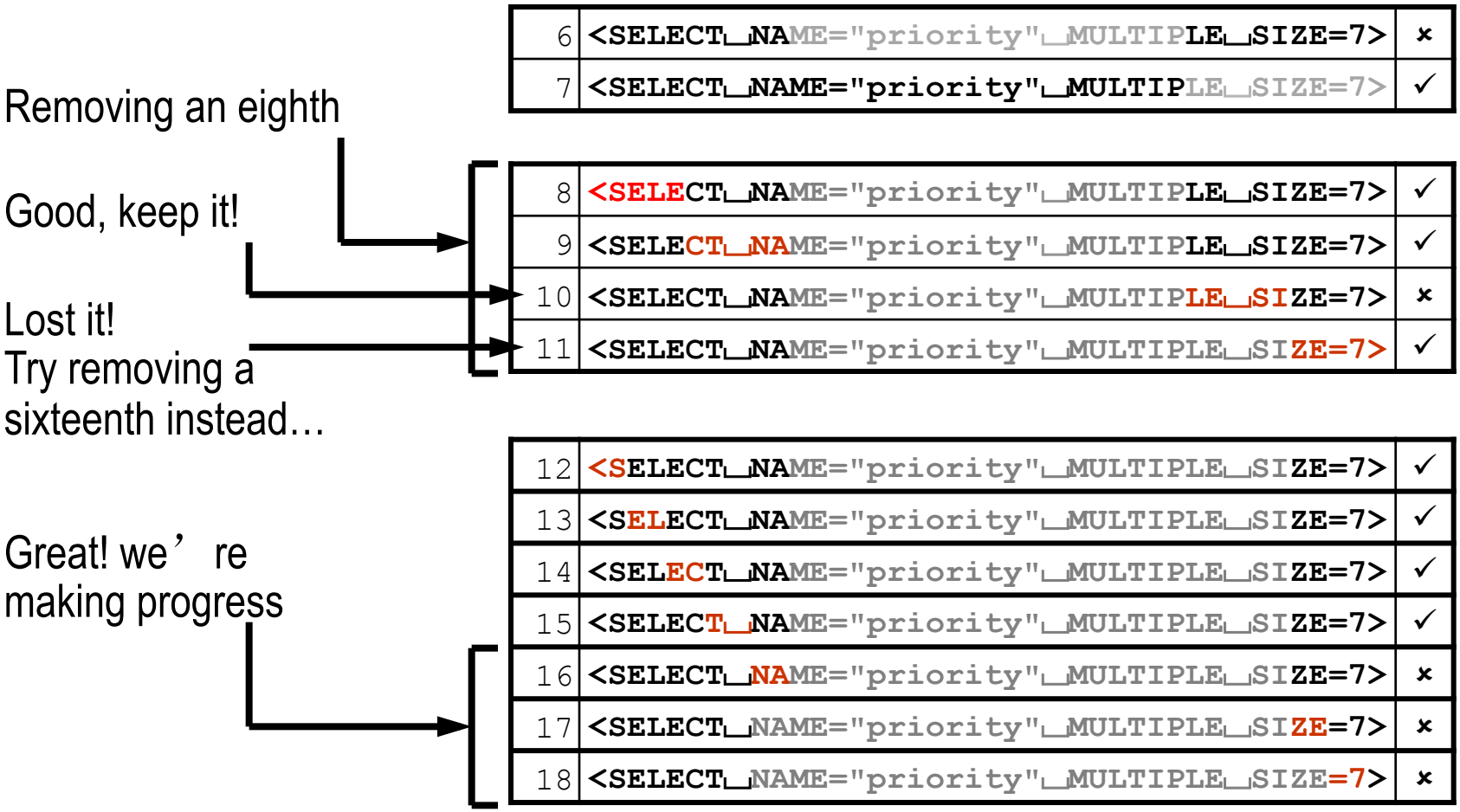
Original failing input



Try removing half…
Now everything passes, we've lost the error inducing input!

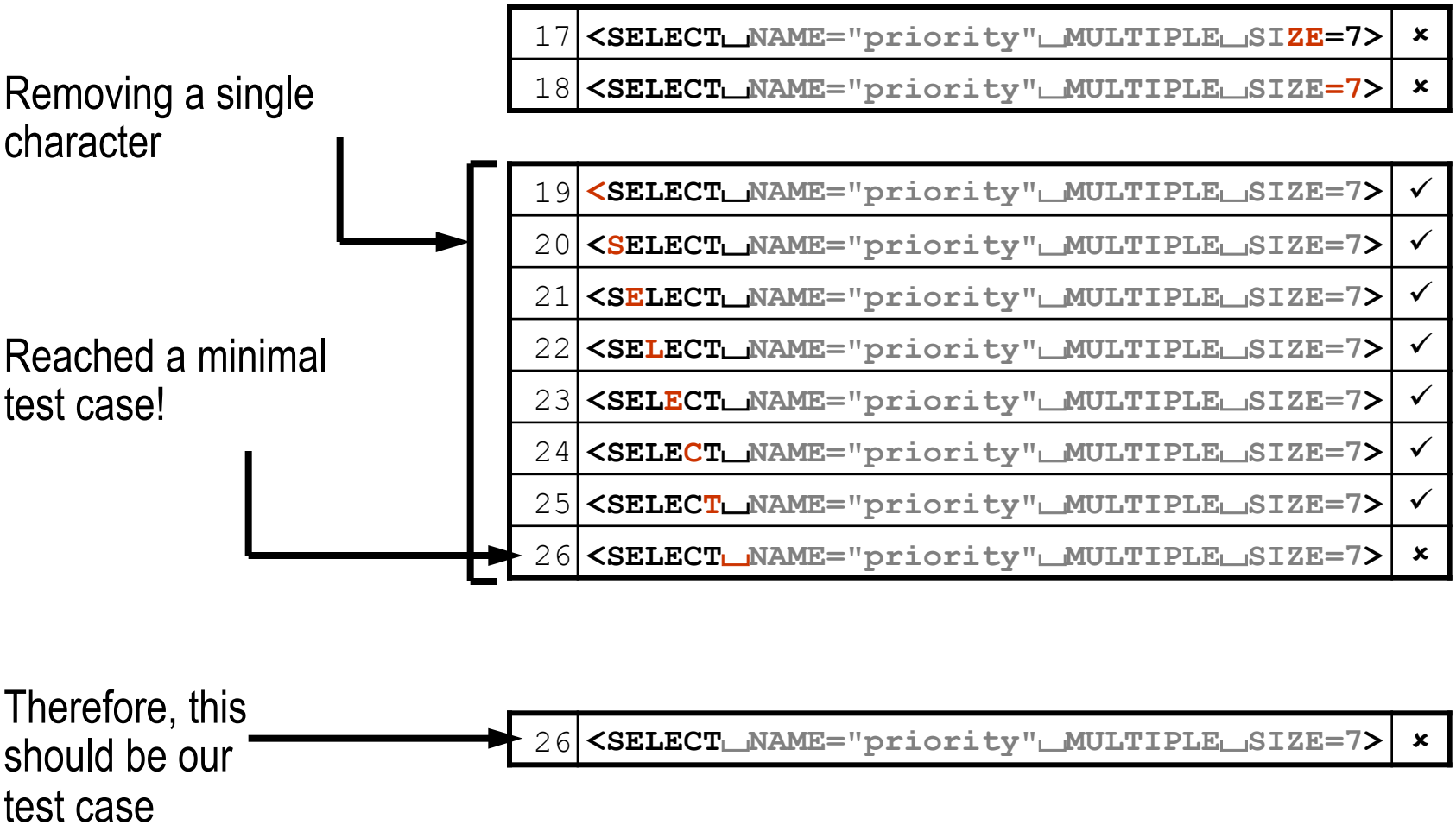
How to minimize a test case?

How to minimize a test case?



OK, now let's see if removing single characters helps us reduce it even more

How to minimize a test case?



Testing for Changes

- The execution of a program is determined by a a number of *circumstances*
  - The program code
  - Data from storage or input devices
  - The program's environment
  - The specific hardware
  - and so on

- The difference between a successful and a faulty run can be viewed as a set of changes on program inputs.

The change that Causes a Failure

- Denote the set of possible configurations of circumstances by R.
- Each $r \in R$ determines a specific program run.
- This r could be
  - a failing run, denoted by $r_{\times}$
  - a passing run, denoted by $r_{\checkmark}$
- Given a specific $r_{\times}$
  - We focus on the difference between $r_{\times}$ and some $r_{\checkmark} \in R$ that works
  - This difference is the change which causes the failure
  - The smaller this change, the better it qualifies as a failure cause

The change that Causes a Failure

- Formally, the difference between $r_{✓}$ and $r_{✗}$ is expressed as a mapping δ which changes the circumstances of a program run

Definition 1 (Change).
A *change* δ is a mapping $\delta : R \rightarrow R$.
The set of changes is $C = R \times R$.
The relevant change between two runs $r_{✓}, r_{✗} \in R$ is a change $\delta \in C$ s.t. $\delta(r_{✓}) = r_{✗}$.

- The exact definition of δ is problem specific
  - In the Mozilla example, applying δ means to expand a trivial (empty) HTML input to the full failure-inducing HTML page.

Decomposing Changes

- We assume that the relevant change δ can be *decomposed* into a number of elementary changes $\delta_1,..., \delta_n$.
- In general, this can be an atomic decomposition
  - Changes that can no further be decomposed

Definition 2 (Composition of changes).
The change composition $\circ : C \rightarrow C \times C$ is defined as
$(\delta_i \circ \delta_j)(r) = \delta_i(\delta_j(r))$

Test Cases and Tests

- According to the POSIX 1003.3 standard for testing frameworks, we distinguish three test outcomes:
  - The test *succeeds* (PASS, written here as ✓)
  - The test has produced the *failure* it was intended to capture (FAIL, written here as ✗)
  - The test produced *indeterminate results* (UNRESOLVED, written as ?)

Definition 3 (*rtest*).
The function $rtest : R \rightarrow \{✓, ✗, ?\}$ determines for a program run $r \in R$ whether some specific failure occurs (✓) or not (✗) or whether the test is unresolved (?).

Axiom 4 (Passing and failing run).
$rtest(r_✓) = ✓$ and $rtest(r_✗) = ✗$ hold.

Test Cases and Tests

- We identify each run by the set of changes being applied to $r_{\checkmark}$
  - We define $c_{\checkmark}$ as the empty set $\varnothing$ which identifies $r_{\checkmark}$ (no changes applied)
  - The set of all changes $c_{\times} = \{\delta_1, \delta_2, ..., \delta_n\}$ identifies $r_{\times} = (\delta_1 \circ \delta_2 \circ ... \circ \delta_n)(r_{\checkmark})$

Definition 5 (Test case).
A subset $c \subseteq c_{\times}$ is called a test case.

Test Cases and Tests

Definition 6 (*test*).
The function $test : 2^{\times} \rightarrow \{\checkmark, \times, ?\}$ is defined as follows:
Let $c \subseteq c_{\times}$ be a test case with $c_{\times} = \{\delta_1, \delta_2, ..., \delta_n\}$. Then
$test(c) = rtest((\delta_1 \circ \delta_2 \circ ... \circ \delta_n)(r_1))$
holds.

Corollary 7 (Passing and failing test cases).
The following holds:

$test(c_{\checkmark}) = test(\emptyset) = \checkmark$	("passing test case")
$test(c_{\times}) = test(\{\delta_1, \delta_2, ..., \delta_n\}) = \times$	("failing test case")

Minimizing Test Cases

Minimal Test Cases

- If a test case $c \subseteq c_{\times}$ is a minimum, no other smaller subset of $c_{\times}$ causes a failure

Definition 8 (Global minimum).
A set $c \subseteq c_{\times}$ is called the *global minimum* of $c_{\times}$ if:
$\forall c' \subseteq c_{\times} \cdot (|c'| < |c| \Rightarrow test(c') \neq \times)$
holds.

- But we don't want to have to test all $2^{|c_{\times}|}$ of $c_{\times}$
- So we'll settle for a *local* minimum
  - A test case is *minimal* if none of its subsets causes a failure

Definition 9 (Local minimum).
A test case $c \subseteq c_{\times}$ is a *local minimum* of $c_{\times}$ or *minimal* if:
$\forall c' \subset c \cdot (test(c') \neq \times)$
holds.

Minimal Test Cases

- Thus, if a test case c is minimal
  - It is not necessarily the smallest test case (there may be a different global minimum)
  - But each element of c is relevant in producing the failure
    - Nothing can be removed without making the failure disappear
- However, determining that c is minimal still requires $2^{|c|}$ tests
- We can use an approximation instead
  - It is possible that removing several changes at once might make a test case smaller
  - But we'll only check if this is so when we remove up to n changes

Minimal Test Cases

- We define n-minimality: removing any combination of up to n changes, causes the failure to disappear
- We're actually most interested in 1-minimal test cases
  - When removing any single change causes the failure to disappear
  - Removing two or more changes at once may result in an even smaller, still failing test case
    - But every single change on its own is significant in reproducing the failure

Definition 10 (n-minimal test case).
A test case $c \subseteq c_{\times}$ is n-minimal if:
$\forall c' \subset c \cdot (|c| - |c'| \leq n \Rightarrow test(c') \neq \times)$ holds.
Consequently, c is 1-minimal if $\forall \delta_i \in c \cdot (test(c - \{\delta_i\}) \neq \times)$ holds.

The Delta Debugging Algorithm

- We partition a given test case $c_\times$ into subsets
  - Suppose we have n subsets $\Delta_1,\ldots,\Delta_n$
  - We test
    - each Δ_i and
    - its complement $\nabla_i = c_\times - \Delta_i$

The Delta Debugging Algorithm

- Testing each Δ_i and its complement, we have four possible outcomes
  - Reduce to subset
    - If testing any Δ_i fails, it will be a smaller test case
    - Continue reducing Δ_i with $n = 2$ subsets
  - Reduce to complement
    - If testing any $\nabla_i = c_\times - \Delta_i$ fails, it will be a smaller test case
    - Continue reducing ∇_i with $n - 1$ subsets
      - Why $n - 1$ subsets and not $n = 2$ subsets? (Maintain granularity!)
  - Double the granularity
  - Done

The Delta Debugging Algorithm

Minimizing Delta Debugging Algorithm

Let *test* and $c_\times$ be given such that $test(\emptyset) = \checkmark \wedge test(c_\times) = \times$ hold.
The goal is to find $c'_\times = ddmin(c_\times)$ such that $c'_\times \subseteq c_\times$, $test(c'_\times) = \times$, and $c'_\times$ is 1-minimal.
The *minimizing Delta Debugging algorithm* $ddmin(c)$ is

$$ddmin(c_\times) = ddmin_2(c_\times, 2) \quad \text{where}$$

$$ddmin_2(c'_\times, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\Delta_i) = \times \text{ (“reduce to subset”)} \\ ddmin_2(\nabla_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = \times \text{ (“reduce to complement”)} \\ ddmin_2(c'_\times, \min(|c'_\times|, 2n)) & \text{else if } n < |c'_\times| \text{ (“increase granularity”)} \\ c'_\times & \text{otherwise (“done”).} \end{cases}$$

where $\nabla_i = c'_\times - \Delta_i$, $c'_\times = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_\times|/n$ holds.
The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c'_\times) = \times \wedge n \leq |c'_\times|$.

Example

- Consider the following minimal test case which consists of the changes δ_1, δ_7, and δ_8
  - Any test case that includes only a subset of these changes results in an unresolved test outcome
  - A test case that includes none of these changes passes the test
- We first partition the set of changes in two halves
  - none of them passes the test

1	$\Delta_1 = \nabla_2$	δ_1	δ_2	δ_3	δ_4	·	·	·	·	?
2	$\Delta_2 = \nabla_1$	·	·	·	·	δ_5	δ_6	δ_7	δ_8	?

Example

- We continue with granularity increased to four subsets

3	Δ_1	δ_1	δ_2	·	·	·	·	·	·	?
4	Δ_2	·	·	δ_3	δ_4	·	·	·	·	✓
5	Δ_3	·	·	·	·	δ_5	δ_6	·	·	✓
6	Δ_4	·	·	·	·	·	·	δ_7	δ_8	?

- When testing the complements, the set ∇_2 fails, thus removing changes δ_3 and δ_4

7	∇_1	·	·	δ_3	δ_4	δ_5	δ_6	δ_7	δ_8	?
8	∇_2	δ_1	δ_2	·	·	δ_5	δ_6	δ_7	δ_8	✗

- We continue with splitting ∇_2 into three subsets

Example

- Steps 9 to 11 have already been carried out and need not be repeated (marked with *)

9	Δ_1	δ_1	δ_2	·	·	·	·	·	·	?*
10	Δ_2	·	·	·	·	δ_5	δ_6	·	·	✓*
11	Δ_3	·	·	·	·	·	·	δ_7	δ_8	✓*

- When testing ∇_2, changed δ_5 and δ_6 can be eliminated

12	∇_1	·	·	·	·	δ_5	δ_6	δ_7	δ_8	?
13	∇_2	δ_1	δ_2	·	·	·	·	δ_7	δ_8	×

- We reduce to ∇_2 and continue with two subsets

Example

14	$\Delta_1 = \nabla_2$	δ_1	δ_2	·	·	·	·	·	·	?*
15	$\Delta_2 = \nabla_1$	·	·	·	·	·	·	δ_7	δ_8	?*

- We increase granularity to four subsets and test each

16	Δ_1	δ_1	·	·	·	·	·	·	·	?
17	Δ_2	·	δ_2	·	·	·	·	·	·	✓
18	Δ_3	·	·	·	·	·	·	δ_7	·	?
19	Δ_4	·	·	·	·	·	·	·	δ_8	?

- Testing the complements shows the we can eliminate δ_2

20	∇_1	·	δ_2	·	·	·	·	δ_7	δ_8	?
21	∇_2	δ_1	·	·	·	·	·	δ_7	δ_8	✗

Example

- The next steps show that none of the remaining changes δ_1, δ_7, and δ_8 can be eliminated

22	Δ_1	δ_1	·	·	·	·	·	·	·	?*
23	Δ_2	·	·	·	·	·	·	δ_7	·	?*
24	Δ_3	·	·	·	·	·	·	·	δ_8	?*
25	∇_1	·	·	·	·	·	·	δ_7	δ_8	?
26	∇_2	δ_1	·	·	·	·	·	·	δ_8	?
27	∇_3	δ_1	·	·	·	·	·	δ_7	·	?

- To minimize this test case, a total of 19 different tests was required

Case Studies

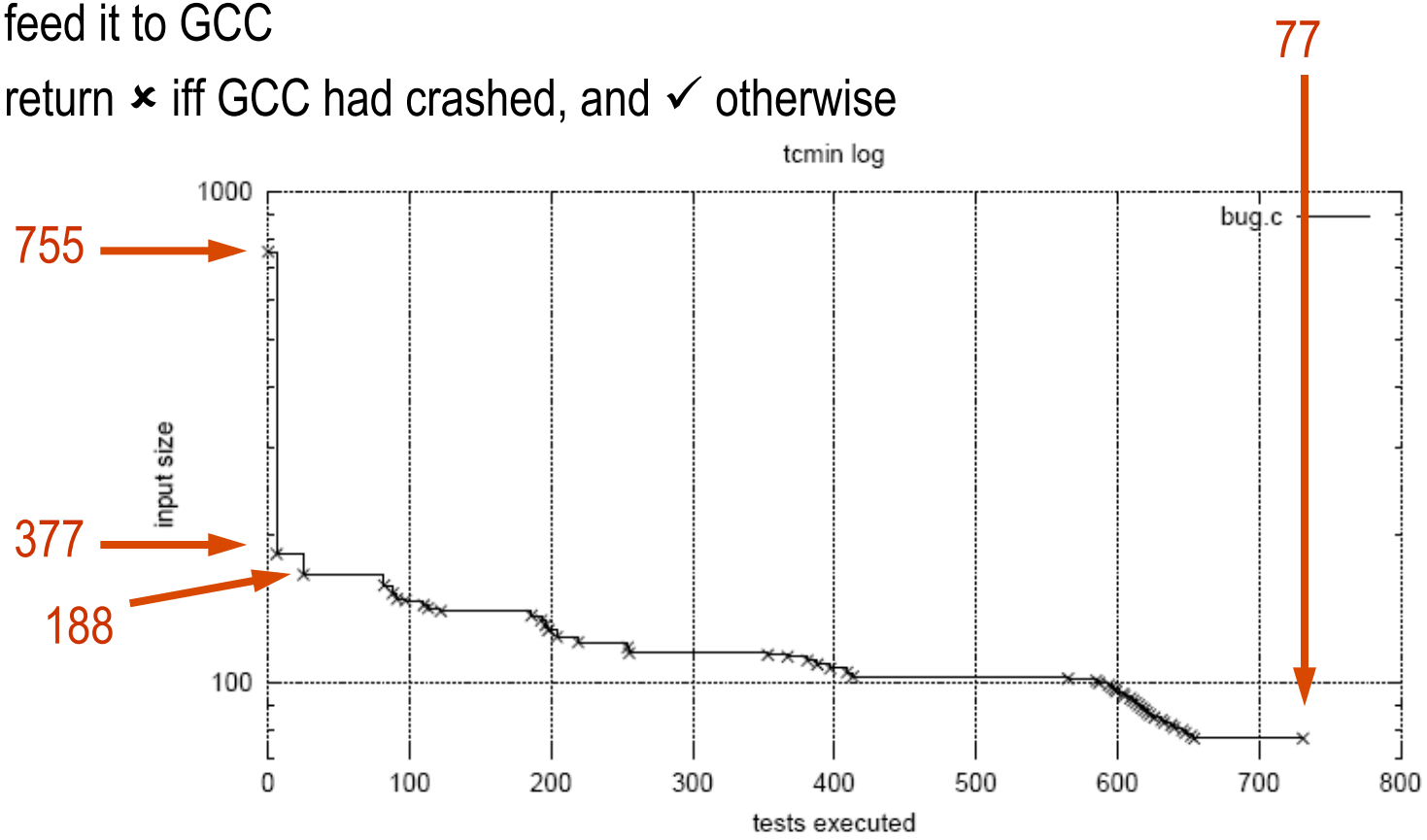
The GNU C Compiler

- This program (**`bug.c`**) causes GCC 2.95.2 to crash when optimization is enabled
- In the case of GCC, a passing program run is the empty input
- The changes are modeled as the insertion of a single character
  - $r_{\checkmark}$ is running GCC with an empty input
  - $r_{\times}$ means running GCC with **`bug.c`**
  - each change δ_i inserts the ith character of **`bug.c`**

```
#define SIZE 20

double mult(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];
}

void copy(double to[], double from[], int count)
{
  int n = (count + 7) / 8;
  switch (count % 8) do {
    case 0: *to++ = *from++;
    case 7: *to++ = *from++;
    case 6: *to++ = *from++;
    case 5: *to++ = *from++;
    case 4: *to++ = *from++;
    case 3: *to++ = *from++;
    case 2: *to++ = *from++;
    case 1: *to++ = *from++;
  } while (--n > 0);
  return mult(to, 2);
}

int main(int argc, char *argv[])
{
  double x[SIZE], y[SIZE];
  double *px = x;

  while (px < x + SIZE)
    *px++ = (px - x) * (SIZE + 1.0);
  return copy(y, x, SIZE);
}
```

The GNU C Compiler

- The test procedure would
  - create the appropriate subset of **`bug.c`**
  - feed it to GCC
  - return ✖ iff GCC had crashed, and ✓ otherwise



The GNU C Compiler

- The minimized code is

```
t(double z[],int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*(z[0]+0);}return z[n];}
```

- The test case is 1-minimal
  - No single character can be removed without removing the failure

- So where could the bug be?
  - We already know it is related to optimization
  - If we remove the `-O` option to turn off optimization, the failure disappears

The GNU C Compiler

- The GCC documentation lists 31 options to control optimization on Linux

-ffloat-store	*-fno-default-inline*	*-fno-defer-pop*
-fforce-mem	*-fforce-addr*	*-fomit-frame-pointer*
-fno-inline	*-finline-functions*	*-fkeep-inline-functions*
-fkeep-static-consts	*-fno-function-cse*	*-ffast-math*
-fstrength-reduce	*-fthread-jumps*	*-fcse-follow-jumps*
-fcse-skip-blocks	*-frerun-cse-after-loop*	*-frerun-loop-opt*
-fgcse	*-fexpensive-optimizations*	*-fschedule-insns*
-fschedule-insns2	*-ffunction-sections*	*-fdata-sections*
-fcaller-saves	*-funroll-loops*	*-funroll-all-loops*
-fmove-all-movables	*-freduce-all-givs*	*-fno-peephole*
-fstrict-aliasing		

- It turns out that applying *all* of these options causes the failure to disappear
  - Some option(s) prevent the failure

The GNU C Compiler

- We can use test case minimization in order to find the preventing option(s)
  - Each δ_i stands for removing a GCC option
  - Having all δ_i applied means to run GCC with no option (failing)
  - Having no δ_i applied means to run GCC with all options (passing)

- After seven tests, the single option `-ffast-math` is found which prevents the failure
  - Again after seven tests, it turn out that `-fforce-addr` also prevents the failure
  - Further examination shows that no other option prevents the failure

The GNU C Compiler

- So, this is what we can send to the GCC maintainers
  - The minimal test case
  - “The failure only occurs with optimization”
  - “`-ffast-math` and `-fforce-addr` prevent the failure”

Conclusion

- Delta debugging can simplify failure-inducing inputs.
- A general, automated, and black-box approach.
  - CReduce, Berkley Delta
  - Compiler Validation via Equivalence Modulo Inputs (PLDI'14)
  - HDD: Hierarchical Delta Debugging (ICSE'06)
- Further, it can isolate failure-inducing difference.

Thank you!

Any Questions?

Minimizing Fuzz

- In a classical experiment by Miller et al. several UNIX utilities were fed with fuzz input – a large number of random characters
  - The studies showed that in the worst case 40% of the basic programs crashed or went into infinite loops
- We would like to use the *ddmin* algorithm to minimize the fuzz input sequences
- We examine the following six UNIX utilities
  - `NROFF` (format documents for display)
  - `TROFF` (format documents for typesetter)
  - `FLEX` (fast lexical analyzer generator)
  - `CRTPLOT` (graphics filter for various plotters)
  - `UL` (underlining filter)
  - `UNITS` (convert quantities)

Minimizing Fuzz

- The following table summarizes the characteristics of the different fuzz inputs used

Name	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}
Character range	all				printable				all				printable			
NUL characters	yes				yes				no				no			
Input size	10^3	10^4	10^5	10^6	10^3	10^4	10^5	10^6	10^3	10^4	10^5	10^6	10^3	10^4	10^5	10^6

Name	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}
NROFF	$\times^S$	$\times^S$	$\times^S$	$\times^S$	–	$\times^A$	$\times^A$	$\times^A$	$\times^S$	$\times^S$	$\times^S$	$\times^S$	–	–	–	–
TROFF	–	$\times^S$	$\times^S$	$\times^S$	–	$\times^A$	$\times^A$	$\times^S$	–	–	$\times^S$	$\times^S$	–	–	–	–
FLEX	–	–	–	–	–	–	–	–	–	–	–	–	–	$\times^S$	$\times^S$	$\times^S$
CRTPLOT	$\times^S$	–	–	–	$\times^S$	–	–	–	$\times^S$	–	–	–	$\times^S$	$\times^S$	$\times^S$	$\times^S$
UL	–	–	–	–	$\times^S$	$\times^S$	$\times^S$	$\times^S$	–	–	–	–	$\times^S$	$\times^S$	$\times^S$	$\times^S$
UNITS	–	$\times^S$	$\times^S$	$\times^S$	–	–	–	–	–	–	$\times^S$	$\times^S$	–	–	–	–

$\times^S$ = segmentation fault, $\times^A$ = arithmetic exception

Minimizing Fuzz

- Out of the $6 \times 16 = 96$ test runs, the utilities crashed 42 times (43%)
- We apply our algorithm to minimize the failure-inducing fuzz input
  - Our *test* function returns ✗ if the input made the program crash
  - or ✓ otherwise