SOFT620020.01
Advanced Software Engineering

Bihuan Chen, Pre-Tenure Assoc. Prof.

bhchen@fudan.edu.cn

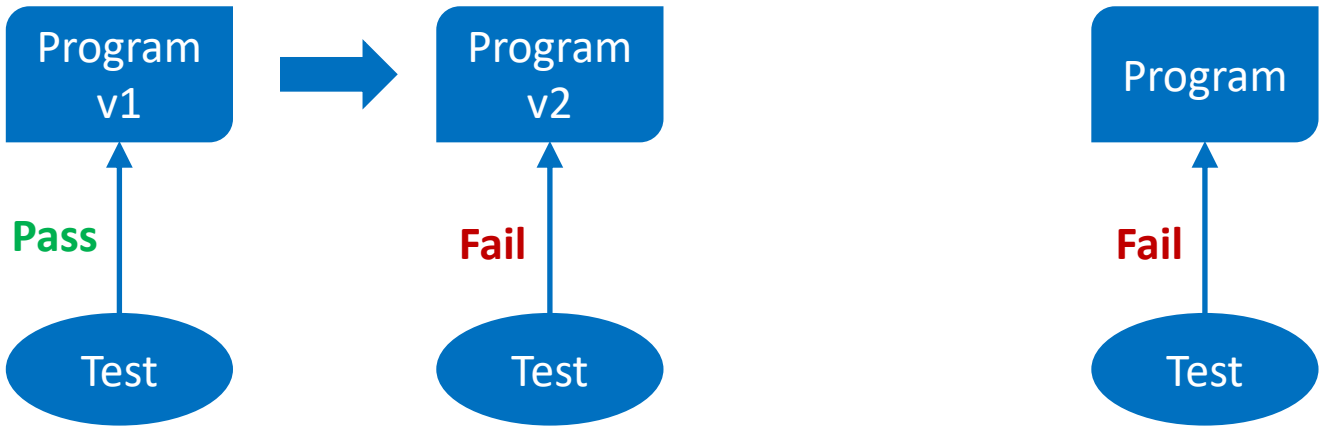
https://chenbihuan.github.io

Course Outline

Date	Topic
Sep. 09	Introduction
Sep. 16	Testing Overview
Sep. 23	Guided Random Testing
Sep. 30	Search-Based Testing
Oct. 12	Performance Analysis
Oct. 14	Presentation 1
Oct. 21	Security Testing
Oct. 28	Compiler Testing

Date	Topic
Nov. 04	Mobile Testing
Nov. 11	**Delta Debugging**
Nov. 18	Presentation 2
Nov. 25	Bug Localization
Dec. 02	Automatic Repair
Dec. 09	Symbolic Execution
Dec. 16	Big Code Analysis
Dec. 23	Presentation 3

Debugging



Debugging (cont.)

- Question 1: Bug Localization
  - How do we localize the bugs?
- Question 2: Bug Fixing
  - How do we fix the bugs?

Yesterday, My Program Worked. Today, It Does Not. Why?

Andreas Zeller

ESEC/FSE 1999, Citation: 420

Motivation

"The GDB people have done it again. The new release 4.17 of the GNU debugger brings several new features, languages, and platforms, but for some reason, it no longer integrates properly with my graphical front-end DDD: the arguments specified within DDD are not passed to the debugged program. Something has changed within GDB such that it no longer works for me. Something? Between the 4.16 and 4.17 releases, no less than 178,000 lines have changed. How can I isolate the change that caused the failure and make GDB work again?"

Delta Debugging (DD)

- DD determines the causes for program behavior by looking at the differences (deltas) between the old and the new program
  - The smaller the differences are, it works the better
  - The differences are usually too large to trace
  - Can we automatically narrow down the differences?

Regression Containment

```
let O be the original working program;
let N be the new buggy program;
let changes be the sequence of changes from O to N;
while (true) {
    apply first half of the changes on O and get ON;
    if (ON is working) {
        let O := ON;
    }
    else {
        let N := ON
    }
    if (the number of changes from O to N is 1)
        return;
}
```

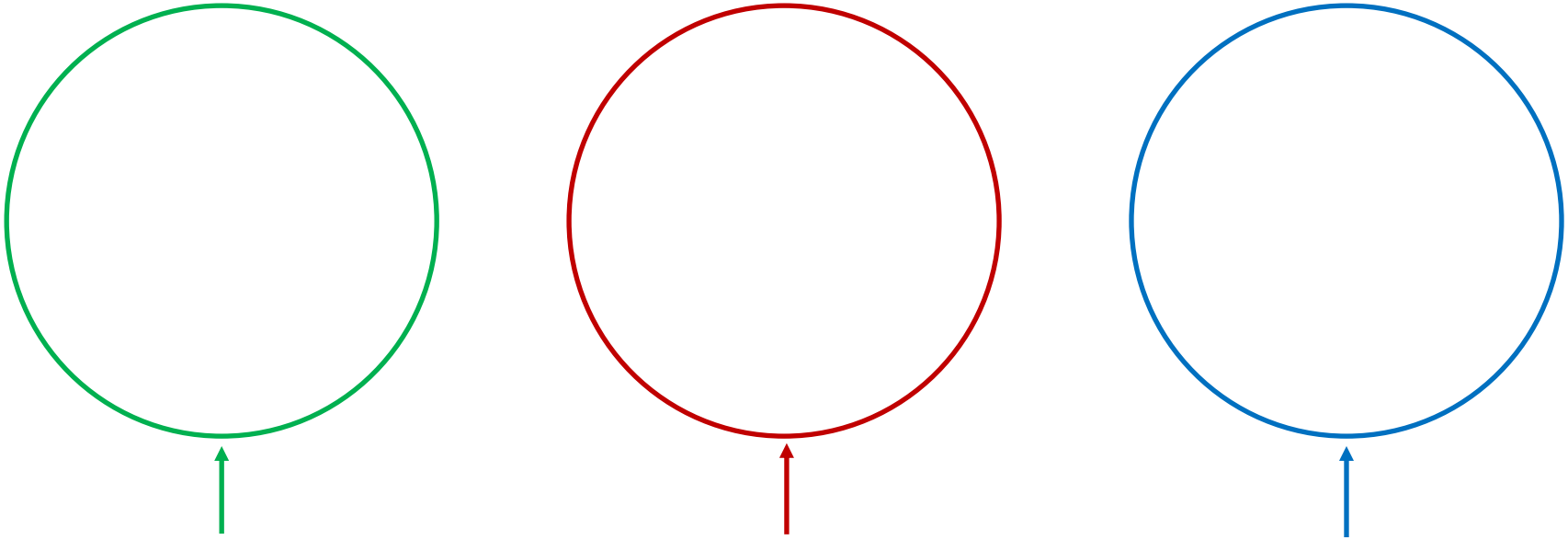
When Will Regression Containment Fails?

- **Interference**
  - There may not be one single change responsible for a failure, but a combination of several changes
- **Inconsistency**
  - Combinations of changes may not result in a testable program (in parallel development)
- **Granularity**
  - A single logical change may affect several hundred or even thousand lines of code, but only a few lines may be responsible for the failure

Subjective of Delta Debugging

- Find **a minimal set of changes** that cause a program to **fail a test case**

Let C= {c_1, c_2, c_3, …, c_n} be the set of changes



O: applied an empty set of changes

ON: applied any subset X of the changes

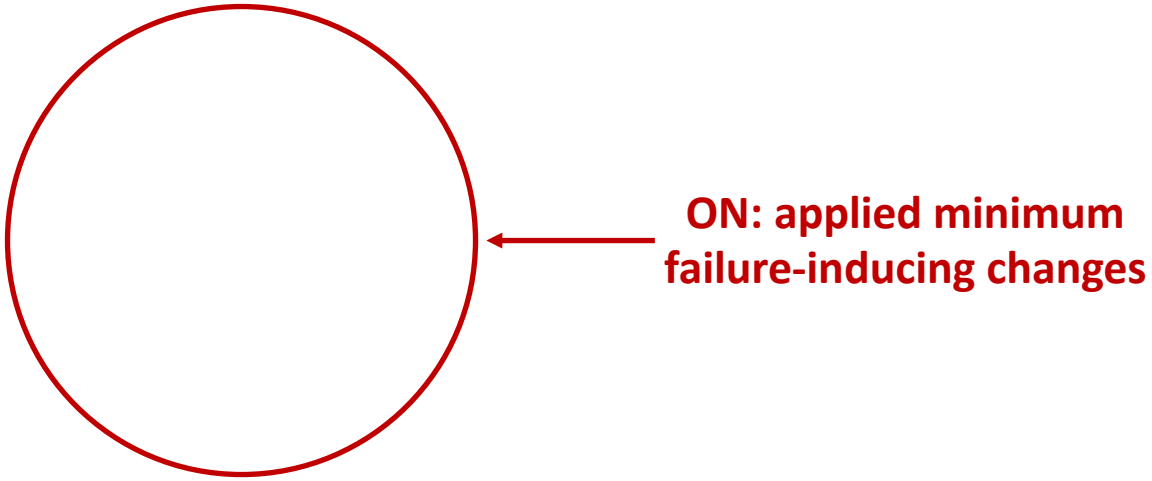
N: applied all changes

Subjective of Delta Debugging (cont.)

- We assume a test produces three testing results
  - **Pass** (✓): the test succeeds
  - **Fail** (×): the test produces the failure
  - **Unresolve** (?): the test produces indeterminate results
- Let **X** be a set of changes and **test(X)** to denote the testing result with the changes in X

Subjective of Delta Debugging (cont.)

- Find a set of changes X such that
  - test(X) != Pass, i.e., failure-inducing set
  - test(Y) != Fail for all Y ⊂ X, i.e., minimum set
  - What is the complexity? **Exponential!**



Simplification Properties/Assumptions

- **Monotony**
  - If test(X) = Fail, test(Y) != Pass for all Y $\supseteq$ X
  - If test(X) = Pass, test(Y) != Fail for all Y $\subseteq$ X
- **Unambiguity**
  - If test(X) = Fail and test(Y) = Fail, test(X $\cap$ Y) != Pass, i.e., a failure is caused by one change set (and not independently by two disjoint sets)
- **Consistency**
  - test(X) != Unresolved for all X

DD Algorithm – The Simplest Case

Searching a single failure-inducing change

DD Algorithm – Interference

Searching two failure-inducing changes

DD Algorithm Formalization

```
algorithm DD(C, R) {
    if (C has one element only) {
        return C;
    }

    partition C into X and Y equally;

    if (test(X ∪ R) = Fail) { // in X
        return DD(X, R)
    }
    if (test(Y ∪ R) = Fail) { // in Y
        return DD(Y, R)
    }

    // interference
    return DD(X, Y ∪ R) ∪ DD(Y, X ∪ R);
}
```

C: a set of changes
R: changes remain to be applied

Initially, dd(C, ∅)

DD Algorithm – Example

Searching eight failure-inducing changes

Assume monotonicity, unambiguity and consistency, DD algorithm always returns the minimum failure-inducing set of changes

Ambiguous

- ~~Unambiguity: if test(X) = Fail and test(Y) = Fail, test(X ∩ Y) != Pass, i.e., a failure is caused by one change set (and not independently by two disjoint sets)~~

- What if it is ambiguous?
  - DD will find one failure-inducing change set
  - Remove this set and apply DD again to find the next

Not Monotone

- ~~Monotony: if test(X) = Fail, test(Y) != Pass for all Y ⊇ X~~

- What if it is not monotone?
  - E.g., test({1,2}) = Fail but test({1,2,3,4}) = Pass
  - DD will find some failure-inducing change set

Inconsistent

- ~~Consistency: test(X) != Unresolved for all X~~

- Potential reasons
  - **Integration failure**: a change may require earlier changes that are not included, or may be conflict with another change but the conflict-resolving change is missing
  - **Construction failure**: although all changes can be applied, the resulting program has syntactical or semantic errors
  - **Execution failure**: the program does not execute correctly, and the test outcome is unresolved

Handling Inconsistency

- Applying less changes can increase the chance of consistency → Partition C into a larger number of subsets
- Extend DD algorithm to work on arbitrary number of subsets
  - **Found**: If test(X) = Fail, X contains a failure-inducing subset
  - **Interference**: If test(X) = test(C–X) = Pass, X and C–X form an interference
  - **Preference**: If test(X) = Unresolved and test(C–X) = Pass, X contains a failure-inducing subset and is preferred, and C–X must remain applied

Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	?	Testing c_1, c_2
2	c_2	.	.	.	.	5	6	7	8	✔	$\Rightarrow$ Prefer c_1
3	c_1	1	2	.	.	5	6	7	8	...	

Handling Inconsistency (cont.)

- There are eight changes. Change 8 is failure-inducing, and changes 2, 3 and 7 only can be applied as a whole

Handling Inconsistency (cont.)

Step	c_i	Configuration								*test*	
11	c_1	1	.	.	.	5	6	.	.	✔	Testing $c_1, \ldots, c_6$
12	c_2	.	2	.	.	5	6	.	.	?	
13	c_3	.	.	3	.	5	6	.	.	?	
14	c_4	.	.	.	4	5	6	.	.	✔	
15	c_5	.	.	.	.	5	6	7	.	?	
16	c_6	.	.	.	.	5	6	.	8	✘	8 is found
Result		.	.	.	.	.	.	.	8		

Changes 1, 4, 5 and 6 have already been identified as not failure-inducing.

If the failure had not been induced by change 8, but by 2, 3, or 7, we would have found it simply by excluding all other changes.

Handling Inconsistency – DD+ Algorithm

```
algorithm DD+(C, R, n) {
    if (C has one element only) { // found
        return C;
    }
    partition C into n sets X1, X2, …, Xn equally;
    if (test(Xi ∪ R) = Fail for some i) { // found in Xi
        return DD+(Xi, R, 2);
    }
    if (test(Xi ∪ R) = Pass and test((C–Xi) ∪ R) = Pass for some i) { // interference
        return DD+(Xi, (C–Xi) ∪ R, 2) ∪ DD+(C–Xi–R, Xi ∪ R, 2);
    }
    if (test(Xi ∪ R) = Unresolved and test((C–Xi) ∪ R) = Pass for some i) { // preference
        return DD+(Xi, (C–Xi) ∪ R, 2);
    }
    let C' = C ∩ {C–Xi–R | test((C–Xi) ∪ R) = Fail};
    if (n < |C|) { // try again
        return DD+(C', R ∪ {Xi | test(Xi ∪ R) = Pass}, min(|C'|, 2n));
    }
    return C';  // nothing left
}
```

Avoiding Inconsistency

- Group related changes
  - E.g., if we know changes 2, 3 and 7 imply each other, we can partition the set into {1,2,3,7} and {4,5,6,8}
  - How to determine whether changes are related: process, location, lexical, syntactic and semantic criteria
- Predict test outcomes
  - If we have evidence about inconsistency, we can predict their test outcomes as unresolved instead of carrying out the test

Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	✔	
2	c_2	.	.	.	.	5	6	7	8	(?)	predicted outcome
3	c_1	1	2	3	4	5	6	.	.	✔	
4	c_2	1	2	3	4	.	.	7	8	(?)	predicted outcome
5	c_1	1	2	3	4	5	6	7	.	✘	7 is found
Result		.	.	.	.	.	.	7	.		

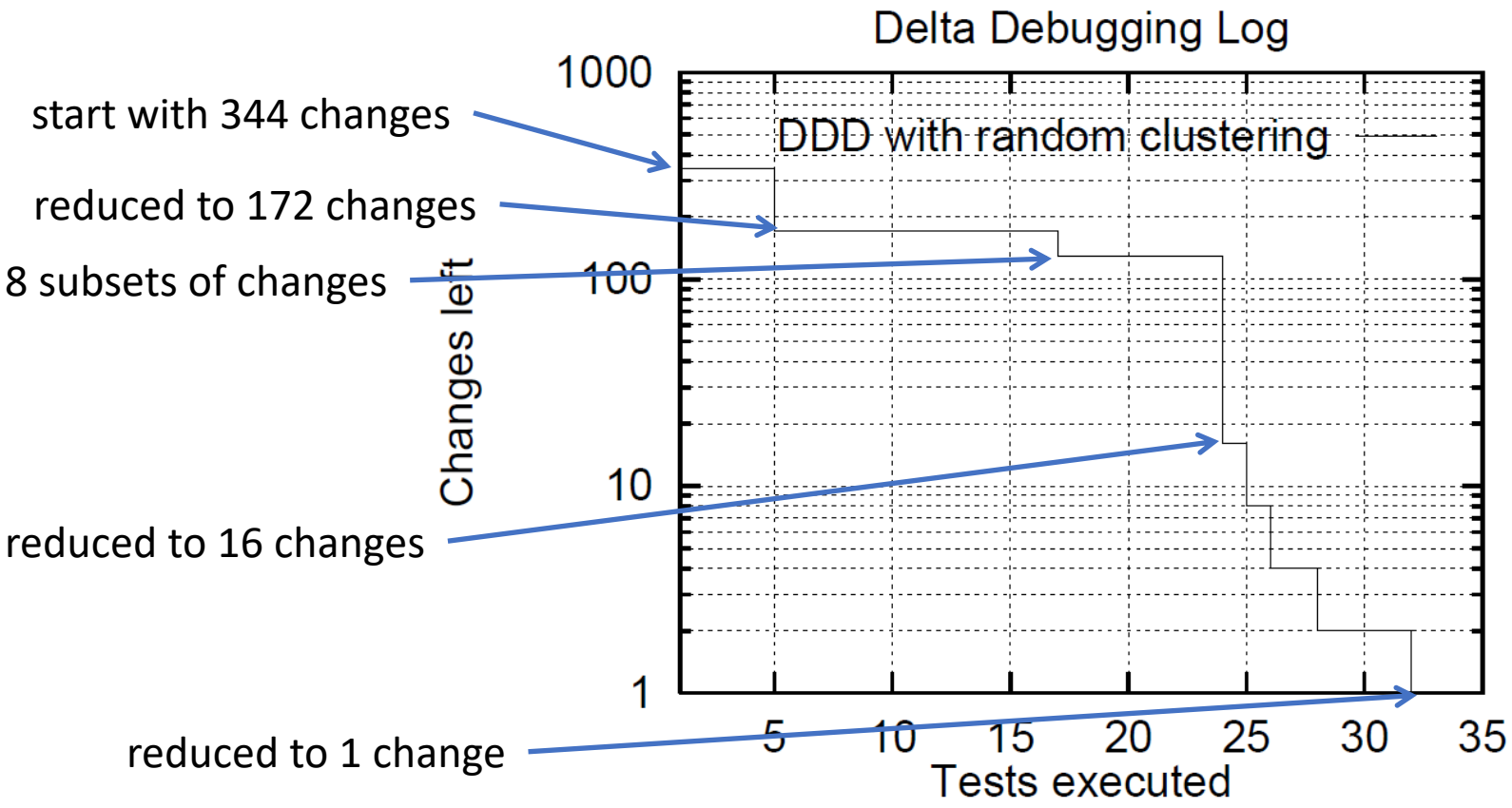
Each change requires all earlier changes to be consistent

Case Study: DDD 3.1.2 Dumps Core

- DDD 3.1.2, released in December, 1998, exhibited a nasty behavioral change: when invoked with the name of a non-existing file, DDD 3.1.2 dumped core, while DDD 3.1.1 simply gave an error message.

- The DDD configuration management archive lists 116 logical changes between the 3.1.1 and 3.1.2 releases. These changes were split into 344 textual changes to the DDD source.

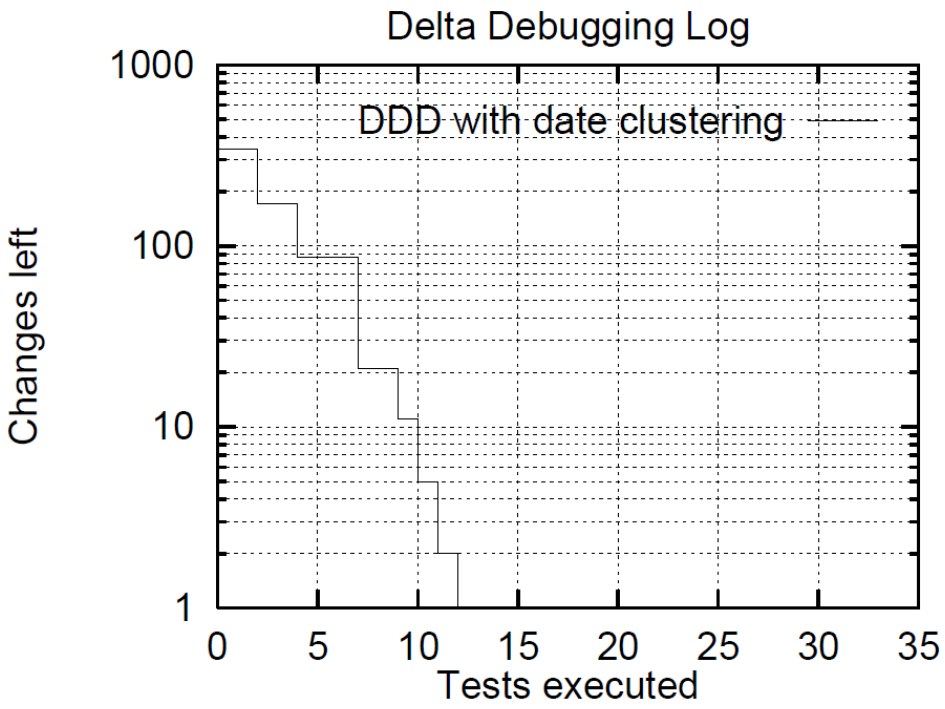
Random Clustering



31 tests in total

Grouping Changes

1. Changes were grouped according to the date they were applied.
2. Each change implied all earlier changes.

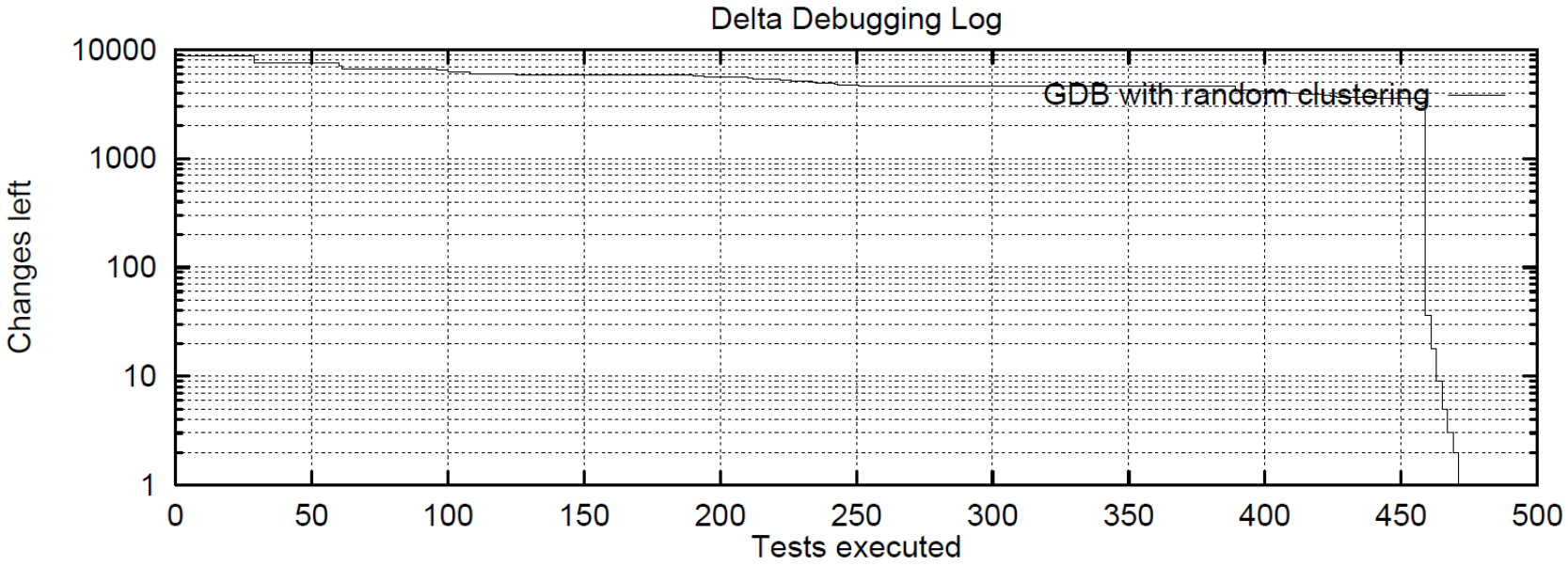


12 tests and 58 minutes

Case Study: GDB 4.17 Does Not Integrate

- 178,000 changed GDB lines, grouped into 8,721 textual changes in the GDB source, with any two textual changes separated by at least two unchanged lines.
- No configuration management archive to obtain change dates, etc.

Random Clustering

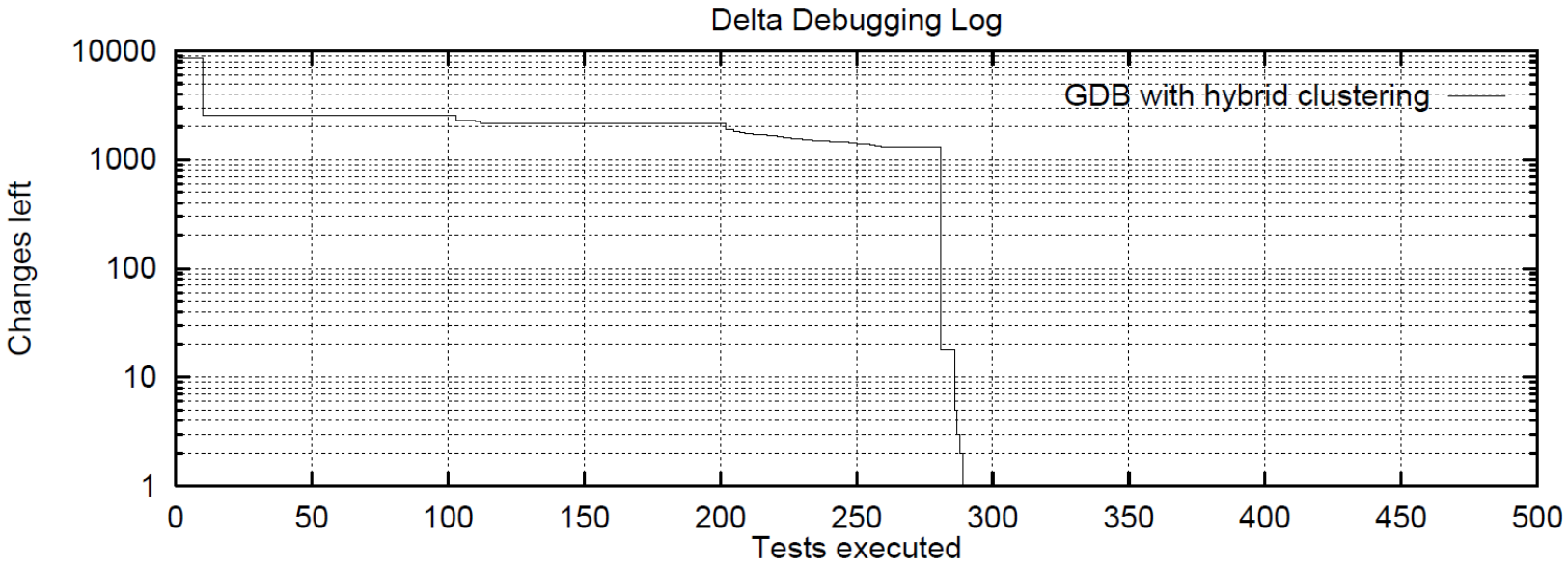


Most of the first 457 tests resulted in unresolved;
At test #458, one subset containing 36 changes resulted in Fail;
Finally, 470 tests in total in 48 hours.

Grouping Changes

- At top-level, changes were grouped according to directories. This was motivated by the observation that several GDB directories contain a separate library whose interface remains more or less consistent across changes.
- Within one directory, changes were grouped according to common files. The idea was to identify compilation units whose interface was consistent with both “yesterday’s” and “today’s” version.
- Within a file, changes were grouped according to common usage of identifiers. This way, we could keep changes together that operated on common variables or functions.
- Finally, a failure resolution loop was added: After a failing construction, scans the error messages for identifiers, adds all changes that reference these identifiers and tries again. This is repeated until construction is possible, or until there are no more changes to add.

Grouping Changes (cont.)



After 9 tests, 2,547 changes left;
After 280 tests, 18 changes left (of two files only);
Finally, 289 tests in total in 20 hours.

Simplifying and Isolating Failure-Inducing Input

Andreas Zeller and Ralf Hildebrandt

TSE 2002, Citation: 808

Motivation

"In July 1999, Bugzilla, the Mozilla bug database, listed more than 370 open bug reports—bug reports that were not even simplified. With this queue growing further, the Mozilla engineers "faced imminent doom". Overwhelmed with work, the Netscape product manager sent out the Mozilla BugAThon call for volunteers: people who would help process bug reports. For 5 bug reports simplified, a volunteer would be invited to the launch party; 20 bugs would earn her or him a T-shirt signed by the grateful engineers."

"Simplifying": turning these bug reports into *minimal test cases*, where every part of the test would be significant in reproducing the failure

Motivation Example

```
<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>
```

Loading this HTML page into Mozilla and printing it causes a segmentation fault

Test Case Simplification

- How to automatically minimize a test case?

- Solution: **Delta Debugging**
  - A test function (with different outputs: Pass, Fail, Unresolved)
    - test(“”) = Pass
    - test(that html page) = Fail
  - A set of primitive changes (which can be composed)
    - insert one character

Minimal Test Cases

- Let **C** be the set of changes containing the primitive change of inserting one character at a position in the input
- Global minimum test case: find a test case $X \subseteq C$ such that
  - Test(X) = Fail
  - Test(Y) != Fail for all $Y \subseteq C$ and $|Y|<|X|$
- Local minimum test case: find a test case $X \subseteq C$ such that
  - Test(X) = Fail
  - Test(Y) != Fail for all $Y \subset X$
- 1-minimal test case: find a test case $X \subseteq C$ such that
  - Test(X) = Fail
  - Test(Y) != Fail for all $Y \subset X$ and $|X| - |Y| = 1$

DDmin Algorithm – Binary Search

Step	Test case									*test*	
1	Δ_1	1	2	3	4	.	.	.	.	?	
2	Δ_2	.	.	.	.	5	6	7	8	✘	
3	Δ_1	.	.	.	.	5	6	.	.	✔	
4	Δ_2	.	.	.	.	.	.	7	8	✘	
5	Δ_1	.	.	.	.	.	.	7	.	✘	Done
Result		.	.	.	.	.	.	7	.		

What if all tests pass or are unresolved?

Testing larger subset will increase the chance that the test fails but have a slower progression

DDmin Algorithm

```
algorithm DDmin(C, n) {
    if (|C| = 1) { //minimum
        return C;
    }
    partition C into X1, X2, …, Xn equally;
    if (test(Xi) = Fail for some i) { // reduce to subset
        return DDmin(Xi, 2);
    }
    if (test(C - Xi) = Fail for some i) { // reduce to complement
        return DDmin(C - Xi, max(n-1, 2));
    }
    if (n < |C|) { // increase granularity
        return DDmin(C, min(|C|, 2n));
    }
    return C;
}
```

DDmin Algorithm – Example

The minimal test case consists of the changes 1, 7 and 8. Any test case that includes a subset of these changes results in an unresolved test outcome; and a test case that includes none of these changes passes the test.

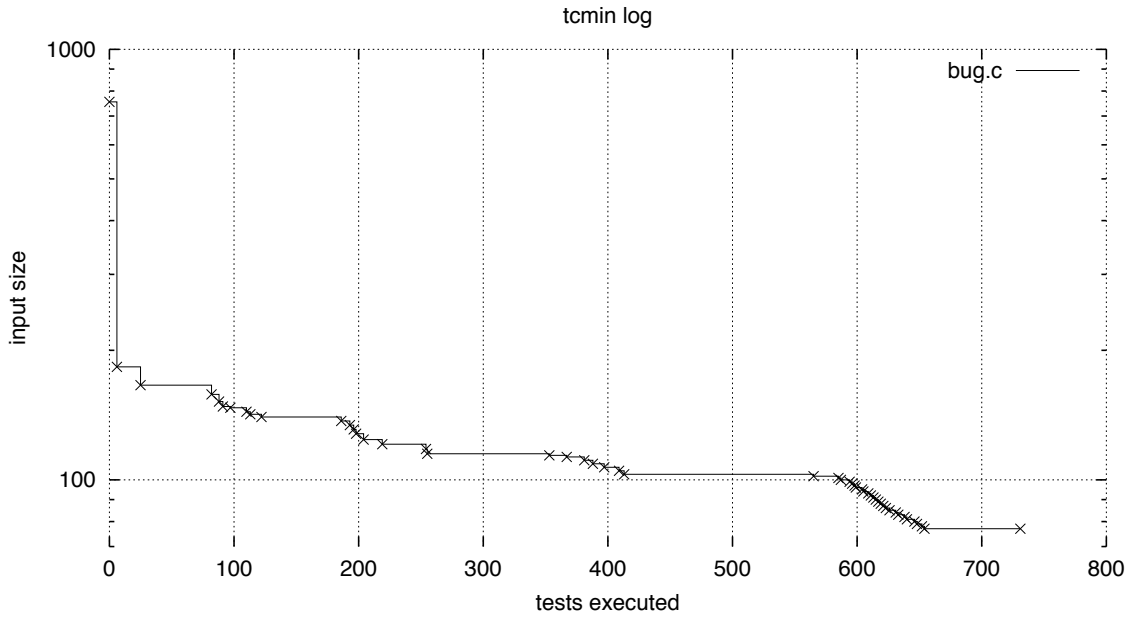
Case Study: GCC

- This program causes GNU C compiler (GCC version 2.95.2 on Intel-Linux with optimization enabled) to crash

- A change is modeled as the insertion of a single character into the program

```
#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);
}
```

Case Study: GCC (cont.)



- After first two tests, the test size is reduced from 755 characters to 377 and 188 characters, i.e., the test only contains the *mult* function
- After 731 more tests (and 34 seconds), the 1-minimal test is found

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

Case Study: GCC (cont.)

- GCC has 31 options

–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		

- When disabling all options, the failure disappears
- When enabling all options, the failure occurs
- Can we find out which option is relevant?
- Yes, simply run DDmin again

Case Study: Mozilla

Reproduction steps of bug #24735 in Mozilla

Ok the following operations cause mozilla to crash consistently on my machine

→ Start mozilla

→ Go to bugzilla.mozilla.org

→ Select search for bug

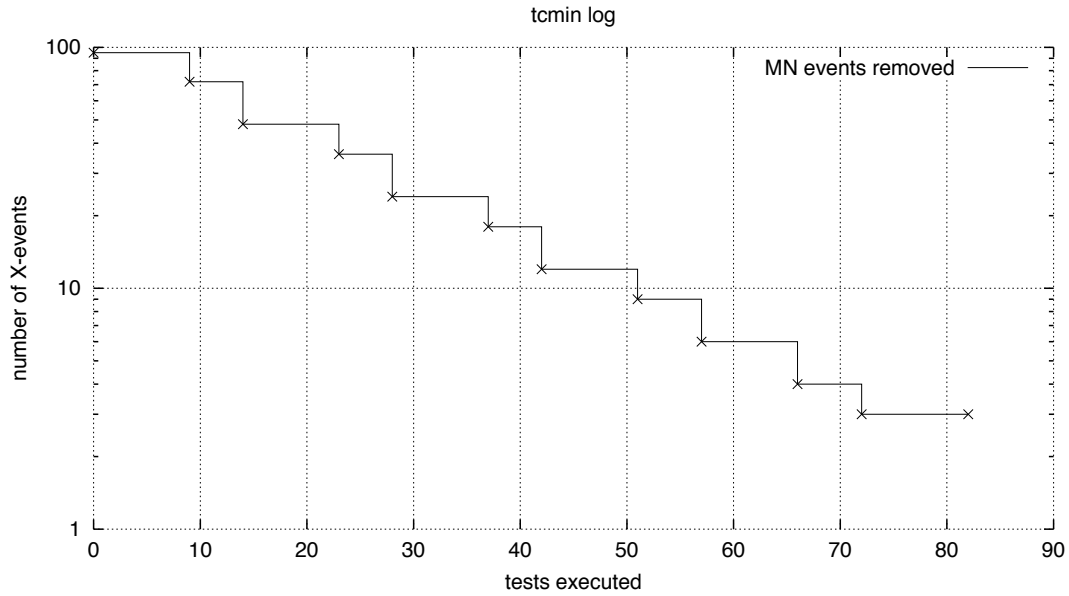
→ Print to file setting the bottom and right margins to .50 (I use the file /var/tmp/netscape.ps)

→ Once it's done printing do the exact same thing again on the same file (/var/tmp/netscape.ps)

→ This causes the browser to crash with a segfault

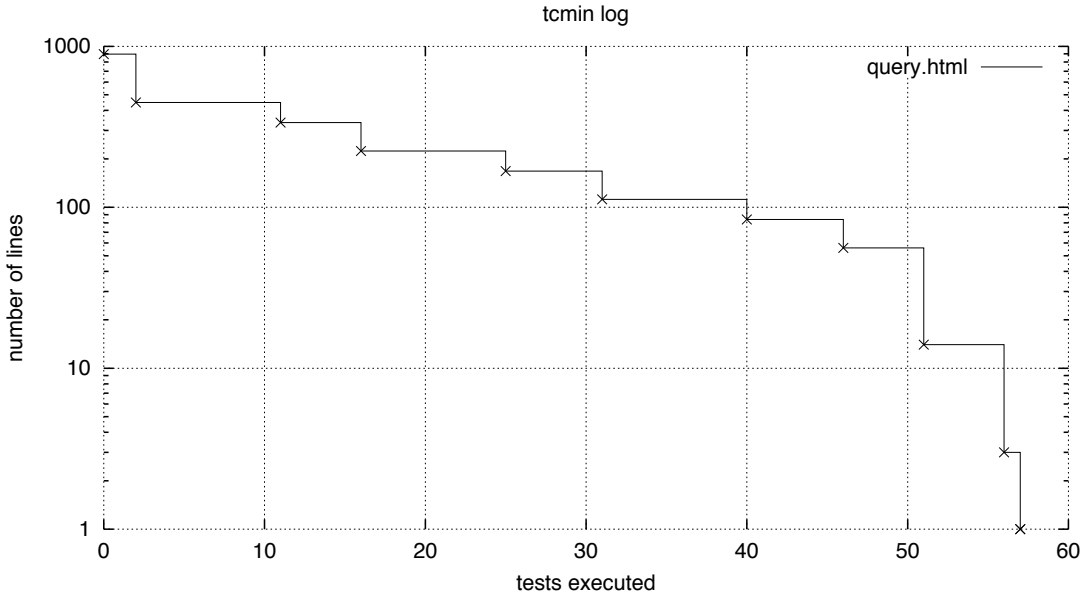
- This Mozilla test input consists of two items
  - Sequence of input events (i.e., 711 recorded XLAB events)
  - HTML code of the erroneous WWW page (i.e., that html page)

Case Study: Mozilla – Minimize User Actions



- After 82 tests (or 21 minutes), 3 out of 95 user actions are left
  - Press the *P* key while holding the *Alt* key (Invoke the *Print* dialog)
  - Press *mouse button 1* on the *Print* button (Arm the *Print* button)
  - Release *mouse button 1* (Start printing)

Case Study: Mozilla – Minimize Code



- Minimize the number of lines in the HTML page
  - 57 tests reduced 896 lines to 1 line

```
<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>
```

Case Study: Mozilla – Minimize Code (cont.)

- Minimize the number of characters
  - 26 tests reduced from 40 characters to 8 characters

```
1  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✘
2  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
3  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
4  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
5  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✘
6  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✘
7  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
8  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
9  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
10 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✘
11 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
12 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
13 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
14 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
15 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
16 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✘
17 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✘
18 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✘
19 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
20 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
21 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
22 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
23 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
24 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
25 <SELECT NAME="priority"␣MULTIPLE␣SIZE=7> ✔
26 <SELECT NAME="priority"␣MULTIPLE␣SIZE=7> ✘
```

Case Study: Fuzzing

(a) Test cases

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

(b) Test outcomes

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	✘^{S}	✘^{S}	✘^{S}	✘^{S}	–	✘^{A}	✘^{A}	✘^{A}	✘^{S}	✘^{S}	✘^{S}	✘^{S}	–	–	–	–
TROFF	–	✘^{S}	✘^{S}	✘^{S}	–	✘^{A}	✘^{A}	✘^{S}	–	–	✘^{S}	✘^{S}	–	–	–	–
FLEX	–	–	–	–	–	–	–	–	–	–	–	–	–	✘^{S}	✘^{S}	✘^{S}
CRTPLOT	✘^{S}	–	–	–	✘^{S}	–	–	–	✘^{S}	–	–	–	✘^{S}	✘^{S}	✘^{S}	✘^{S}
UL	–	–	–	–	✘^{S}	✘^{S}	✘^{S}	✘^{S}	–	–	–	–	✘^{S}	✘^{S}	✘^{S}	✘^{S}
UNITS	–	✘^{S}	✘^{S}	✘^{S}	–	–	–	–	–	–	✘^{S}	✘^{S}	–	–	–	–

"–" = test passed (✔), ✘^{S}= Segmentation Fault, ✘^{A}= Arithmetic Exception

Case Study: Fuzzing (cont.)

(c) Size $|c'_{\times}|$ of minimized input—low precision

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	2	2	2	2	–	7	7	7	2	2	2	2	–	–	–	–
TROFF	–	3	3	3	–	7	7	7	–	–	3	3	–	–	–	–
FLEX	–	–	–	–	–	–	–	–	–	–	–	–	–	2121	2121	2121
CRTPLOT	1	–	–	–	1	–	–	–	1	–	–	–	1	1	1	1
UL	–	–	–	–	516	516	516	516	–	–	–	–	516	516	516	516
UNITS	–	77	77	77	–	–	–	–	–	–	n/a	n/a	–	–	–	–

Test outcomes t_{11} and t_{12} for UNITS could not be reproduced deterministically.

(d) Number of test runs—low precision

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	55	41	60	39	–	156	153	243	17	22	27	54	–	–	–	–
TROFF	–	84	73	100	–	156	153	22493	–	–	50	42	–	–	–	–
FLEX	–	–	–	–	–	–	–	–	–	–	–	–	–	11589	17960	10619
CRTPLOT	15	–	–	–	15	–	–	–	16	–	–	–	14	17	23	24
UL	–	–	–	–	7138	7012	6058	7090	–	–	–	–	2434	3455	3055	2307
UNITS	–	662	623	626	–	–	–	–	–	–	n/a	n/a	–	–	–	–

Case Study: Fuzzing (cont.)

(e) Size $|c'_{\times}|$ of minimized input—high precision

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	60	61	54	60	–	9	9	9	54	54	61	54	–	–	–	–
TROFF	–	393	392	204	–	9	9	9	–	–	73	8	–	–	–	–
FLEX	–	–	–	–	–	–	–	–	–	–	–	–	–	6749	6749	6749
CRTPLOT	1	–	–	–	1	–	–	–	1	–	–	–	1	1	1	1
UL	–	–	–	–	516	516	516	516	–	–	–	–	516	516	516	516
UNITS	–	77	77	77	–	–	–	–	–	–	n/a	n/a	–	–	–	–

(f) Number of test runs—high precision

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	3547	3468	3941	3403	–	150	141	229	4131	4246	5565	2722	–	–	–	–
TROFF	–	43963	39426	10487	–	150	141	229	–	–	2372	1001	–	–	–	–
FLEX	–	–	–	–	–	–	–	–	–	–	–	–	–	37029	34450	37454
CRTPLOT	16	–	–	–	15	–	–	–	16	–	–	–	14	17	23	24
UL	–	–	–	–	7138	7012	6058	7090	–	–	–	–	2434	3455	3055	2307
UNITS	–	684	623	626	–	–	–	–	–	–	n/a	n/a	–	–	–	–

Discussion – Improve DDmin?

- Are all generated test cases meaningful?
- How do we partition the input?
- Which test case to run first?
- What if there are multiple failure-inducing inputs?

Reading Materials

- Andreas Zeller. Yesterday, My Program Worked. Today, It Does Not. Why? In ESEC/FSE, pp. 253-267, 1999.
- Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. In IEEE Transactions on Software Engineering, 28(2), pp. 183-200, 2002.
- Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In ISSTA, pp. 210-220, 2002.
- Andreas Zeller. Isolating cause-effect chains from computer programs. In FSE, pp. 1-10, 2002.
- Holger Cleve and Andreas Zeller. Locating causes of program failures. In ICSE, pp. 342-351, 2005.
- Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In ASE, pp. 263-272, 2005.
- Ghassan Misherghi and Zhendong Su. HDD: hierarchical delta debugging. In ICSE 2006, pp. 142-151, 2006.

Q&A?

Bihuan Chen, Pre-Tenure Assoc. Prof.

bhchen@fudan.edu.cn

https://chenbihuan.github.io