Simplifying and Isolating Failure-Inducing Input

Andreas Zeller, *Member, IEEE Computer Society*, and Ralf Hildebrandt

Abstract—**Given some test case, a program fails. Which circumstances of the test case are responsible for the particular failure? The** *Delta Debugging* **algorithm generalizes and simplifies some failing test case to a** *minimal test case* **that still produces the failure; it also isolates the** *difference* **between a passing and a failing test case.**

In a case study, the Mozilla web browser crashed after 95 user actions. Our prototype implementation automatically simplified the input to 3 relevant user actions. Likewise, it simplified 896 lines of HTML to the single line that caused the failure. The case study required 139 automated test runs, or 35 minutes on a 500 MHz PC.

Index Terms—automated debugging, debugging aids, testing tools, combinatorial testing, diagnostics, tracing.

I. INTRODUCTION

> *Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.*
>
> — Richard Stallman, *Using and Porting* GNU CC

If you browse the Web with Netscape 6, you actually use a variant of *Mozilla*, Netscape's open source web browser project [1]. As a work in progress with big exposure, the Mozilla project receives several dozens of bug reports a day. The first step in processing any bug report is *simplification,* that is, eliminating all details that are irrelevant for producing the failure. Such a simplified bug report not only facilitates debugging, but it also subsumes several other bug reports that only differ in irrelevant details.

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" [2]. Overwhelmed with work, the Netscape product manager sent out the *Mozilla BugAThon call for volunteers* [2]: 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" meant: turning these bug reports into *minimal test cases,* where every part of the input would be significant in reproducing the failure.

As an example, consider the HTML input in Figure 1. Loading this HTML page into Mozilla and printing it causes a segmentation fault. Somewhere in this HTML input is something that makes Mozilla fail—but where? If we were Netscape programmers, what we wanted here is the simplest HTML page that still produces the failure.

Decomposing specific bug reports into simple test cases does not trouble only the Mozilla engineers. The problem arises from generally conflicting issues:

- A *bug report* should be as *specific* as possible, such that the engineer can recreate the context in which the program failed.
- On the other hand, a *test case* should be as *simple* as possible, because a minimal test case implies a most general context.

Thus, a minimal test case not only allows for short problem descriptions and valuable problem insights, but it also subsumes several current and future bug reports.

The striking thing about test case simplification is that no one so far has thought to *automate* this task. Several textbooks and guides about debugging are available that tell how to use binary search in order to isolate the problem—based on the assumption that tests are carried out manually, too. With an automated test, however, we can automate this *simplification of test cases,* and we can automatically *isolate the difference that causes the failure.*

Simplification of test cases. Our *minimizing delta debugging algorithm ddmin* is fed with a failing test case, which it simplifies by successive testing. It stops when a *minimal test case* is reached, where removing any single input entity would cause the failure to disappear.

As an analogon from the real world, consider a *flight test:* an air plane crashes a few seconds after taking off. By repeating the situation over and over again under changed circumstances, we can find out what is relevant and what not. For instance, we may leave away the passenger seats and find that the plane still crashes. We may leave away the coffee machine and the plane still crashes. Eventually, only the relevant "simplified" skeleton remains, including a test pilot, the wings, the runway, the fuel, and the engines. Each part of this skeleton is relevant for reproducing the crash.

In the real world, no one with a sane mind would consider such a way to simplify the circumstances of test flights. However, for *simulations* of flight tests, or, more generally, for arbitrary computer programs, such an approach comes at a far lesser cost. The cost may be so low that we can easily use a large amount of tests just to simplify a test case.

Figure 2 sketches how the *ddmin* procedure minimizes a test case: Starting with the HTML input in Figure 1, the *ddmin* algorithm simplifies the input by testing subsets with removed characters (shown in grey): The test fails (✘) if Mozilla crashes on the given test case and passes (✔) otherwise. After 57 tests, the original 896-line HTML input is reduced to the minimal failing test case `<SELECT>`.[1] Each character in the minimal failing test case is relevant for producing the failure.

Isolating failure-inducing differences. In the case where a *passing test case* exists as well, it is generally more efficient to isolate the *failure-inducing difference* between a failing and a passing test case. This is what the *general Delta Debugging algorithm dd* does. *dd* is a generalization of *ddmin.*

Again, as an analogon, take the flight test example. Now, we try to isolate the difference between the crash and a working

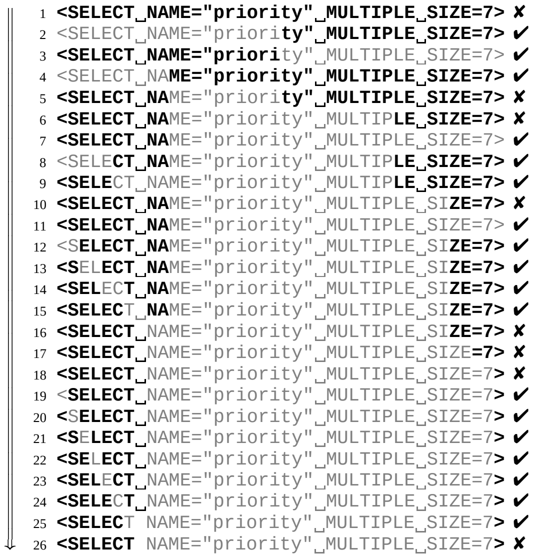
Andreas Zeller is with Universität des Saarlandes, Lehrstuhl für Softwaretechnik, Postfach 15 11 50, 66041 Saarbrücken, Germany. E-mail: zeller@computer.org.

Ralf Hildebrandt is with DeTeLine – Deutsche Telekom Kommunikationsnetze GmbH, Rognitzstrasse 9, 14057 Berlin, Germany. E-mail: ralf_hildebrandt@web.de.

[1] Section IV-B has more details on this example.

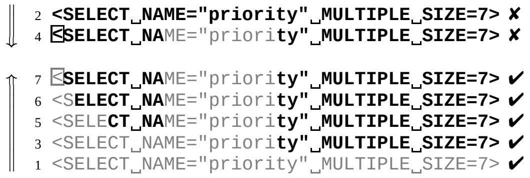
```
<td align=left valign=top>
<SELECT NAME="op_sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows 3.1<OPTION VALUE="Windows 95">Windows 95<OPTION VALUE="Windows
98">Windows 98<OPTION VALUE="Windows ME">Windows ME<OPTION VALUE="Windows 2000">Windows 2000<OPTION VALUE="Windows
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>
```

Fig. 1. Printing this HTML page makes Mozilla crash (excerpt)



```
 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> ✘
```

Fig. 2. Simplifying failure-inducing HTML input



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

7 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
6 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
5 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
3 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
1 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
```

Fig. 3. Isolating a failure-inducing difference

flight. We may find that if we replace the engines of the crashing machine by the engines of a working machine, the change does not matter; consequently, we find this difference to be irrelevant. By reducing differences further and further, we may eventually isolate a piece of metal on the runway that is relevant for the crash—everything else may stay the same, but having this piece of metal on the runway or not induces whether there is a crash or a perfect flight.

Again, nobody wants to crash planes over and over. Figure 3 shows how *dd* works on HTML input: Rather than only minimizing the failing input, *dd* also *maximizes* the passing HTML input until a minimal failure-inducing difference is obtained. In our case, this is the first character < of the failure-inducing `<SELECT>` tag, pinpointed after only seven tests: This one-character difference makes Mozilla fail.

Delta Debugging as a technique for simplifying or isolating failure causes is not limited to HTML input, to character input, nor to program input in general: Delta Debugging can be applied to *all circumstances that in any way affect the program execution.* Delta Debugging is fully automatic: whenever some regression test fails, an additional Delta Debugging run automatically determines the failure-inducing circumstances.

In earlier work [3], we have shown how Delta Debugging is applied to isolate failure-inducing code changes; our current research includes application domains like failure-inducing thread schedules or failure-inducing program statements. In this paper, however, we will concentrate on *program input*.

The remainder of this paper is organized as follows: We begin with formal definitions of passing and failing test cases (Section II). We first introduce the basic *ddmin* algorithm in Section III which simplifies failing test cases. The case studies (Section IV) include GCC, Mozilla, and UNIX utilities subjected to random fuzz input.

In Section V, we extend *ddmin* to *dd* to isolate the difference between a passing and a failing test case. Section VI evaluates *dd* by repeating the GCC and fuzz case studies. Sections VII and VIII close with discussions of related and future work.

II. Testing for Change

Software features that can't be demonstrated by automated tests simply don't exist.

— Kent Beck, *Extreme Programming Explained*

In general, we assume that the execution of a specific program is determined by a number of *circumstances.* These circumstances include the program code, data from storage or input devices, the program's environment, the specific hardware, and so on.

In our context, we are only interested in the *changeable circumstances*—that is, those circumstances whose change may cause a different program behaviour. (In fact, we are even only interested in those circumstances that actually may cause a different test outcome.) These changeable circumstances make up the program's input (in the most general sense). In the remainder of this paper, "circumstances" will always refer to changeable circumstances.

A. The Change that Causes a Failure

Let us denote the set of possible configurations of circumstances by ¨. Each $r \in ¨$ determines a specific program run. Let us assume now a specific run $r_{✘} \in ¨$ that fails.[2] Typically, we do not consider all circumstances of this run as a whole. Instead, we focus on the *difference* between $r_{✘}$ and some run $r_{✔} \in ¨$ that works. This difference is the change which causes the failure, and the smaller this change, the better it qualifies as failure cause.

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 $\delta : ¨ \to ¨$. The set of changes is $+ = ¨^{¨}$. The* relevant change *between two runs $r_{✔}, r_{✘} \in ¨$ is a change $\delta \in +$ such that $\delta(r_{✔}) = r_{✘}$.*

In the remainder of this paper, δ will always stand for the relevant change between the two given program runs $r_{✔}$ and $r_{✘}$. The exact definition of δ and its application is, of course, specific to the given problem and its circumstances. In the Mozilla example sketched in Section I, applying δ means to expand a trivial (empty) HTML input to the full failure-inducing HTML page.

B. Decomposing Changes

We now assume that the relevant change δ can be *decomposed* into a number of elementary changes $\delta_1, \dots, \delta_n$. This decomposition of δ into individual changes δ_i is problem-specific. As an example, think of a DIFF output δ consisting of several individual changes δ_i, each affecting a particular place in the text.

Our approach does not suggest a specific way of decomposing changes. In general, though, we expect the decomposition to follow the structure of the change, which again follows the structure of the circumstances being changed. In the Mozilla example from Section I, there are many ways to decompose the change δ: it may be decomposed into changes adding single characters, or changes adding HTML tags, or changes adding lines, depending on whether we see the input of being composed from characters, tags, or lines. In doubt, an *atomic* decomposition—that is, a decomposition into changes that can no further be decomposed—is the way to go.

To express (de-)composition formally, we write $\delta = \delta_1 \circ \delta_2 \circ \cdots \circ \delta_n$, where the composition $\delta_i \circ \delta_j$ groups two changes δ_i and δ_j into a larger whole:

Definition 2 (Composition of changes) *The* change composition *$\circ : + \times + \to +$ is defined as $(\delta_i \circ \delta_j)(r) = \delta_i\big(\delta_j(r)\big)$.*

We do not assume any particular properties of $\circ$. In practice, $\circ$ is typically realized as a *union* of two change sets δ_i.

C. Test Cases and Tests

To relate program runs to failures, we need a testing function that takes a program run and tests whether it produces the failure. According to the POSIX 1003.3 standard for testing frameworks [4], we distinguish three 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 here as **?**).[3]

Definition 3 (*rtest*) *The function rtest : $¨ \to \{✘, ✔, ?\}$ determines for a program run $r \in ¨$ 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.*

In the remainder of this paper, we shall consider not only $r_{✔}$ and $r_{✘}$, but also several runs that are the product of changes being applied to $r_{✔}$. For convenience, we identify each run by *the set of changes being applied to $r_{✔}$*. That is, we define $c_{✔}$ as the empty set $c_{✔} = \emptyset$ which identifies $r_{✔}$ (no changes applied). The set of all changes $c_{✘} = \{\delta_1, \delta_2, \dots, \delta_n\}$ identifies $r_{✘} = (\delta_1 \circ \delta_2 \circ \cdots \circ \delta_n)(r_{✔})$.

We call the subsets of $c_{✘}$ *test cases:*

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

Test cases are related to program runs by means of the *test* function, which applies the set of changes to $r_{✔}$ and tests the resulting run.

Definition 6 (*test*) *The function test : $2^{c_{✘}} \to \{✘, ✔, ?\}$ is defined as follows: Let $c \subseteq c_{✘}$ be a test case with $c = \{\delta_1, \delta_2, \dots, \delta_n\}$. Then, $test(c) = rtest\big((\delta_1 \circ \delta_2 \circ \cdots \circ \delta_n)(r_{✔})\big)$ holds.*[4]

Using Axiom 4, we can deduce the results of $test(c_{✔})$ and $test(c_{✘})$:

Corollary 7 (Passing and failing test case) *The following holds:*

$$\begin{array}{ll} test(c_{✔}) = test(\emptyset) = ✔ & \text{("passing test case") and} \\ test(c_{✘}) = test\big(\{\delta_1, \delta_2, \dots, \delta_n\}\big) = ✘ & \text{("failing test case").} \end{array}$$

III. Minimizing Test Cases

> *Proceed by binary search. Throw away half the input and see if the output is still wrong; if not, go back to the previous state and discard the other half of the input.*
>
> — Brian Kernighan and Rob Pike,
> *The Practice of Programming*

Let us now model our initial scenario. We have a test case $c_{✔}$ that works fine and a test case $c_{✘}$ that fails. Let us assume that $c_{✔}$ stands for some trivial program run (such as a run on an empty input). Then, minimizing the difference between $c_{✔}$ and $c_{✘}$ becomes *minimizing $c_{✘}$ itself*—that is, *simplification* of $c_{✘}$.

[2] Read $r_{✘}$ and $r_{✔}$ as "r-fail" and "r-pass", respectively.

[3] POSIX 1003.3 also lists UNTESTED and UNSUPPORTED outcomes, which are of no relevance here.

[4] To make the application of change sets unambiguous, *test* must sort the applied changes δ_i in some canonical way.

A. Minimal Test Cases

A test case $c \subseteq c_{\text{✘}}$ being a minimum means that there is no smaller subset of $c_{\text{✘}}$ that causes the test to fail. Formally:

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

In practice, this would be nice to have, but it is practically impossible to compute: Relying on *test* alone to determine the global minimum of $c_{\text{✘}}$ requires testing all $2^{|c_{\text{✘}}|}$ subsets of $c_{\text{✘}}$, which obviously has exponential complexity.[5]

Resorting to the idea of a *local minimum* helps a little. We call a test case *minimal* if none of its subsets causes the test to fail. That is, if a test case c is minimal, there may be some other test case that is even smaller (i.e. a global minimum), but at least we know that each element of c is relevant in producing the failure—nothing can be removed without making the failure disappear.

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

This is what we want: a failing test case whose elements are all significant. However, determining that a test case c is a local minimum still requires $2^{|c|} - 2$ tests.

What we can determine, however, is an *approximation*—for instance, a test case where removing a small set of changes is still significant in producing the failure, but we do not check whether removing several changes at once might make the test case even smaller. Formally, we define this property as *n-minimality:* removing any combination of up to n changes causes the failure to disappear. If c is $|c|$-minimal, then c is minimal in the sense of Definition 9.

The approximation which interests us most is *1-minimality*. A failing test case c composed of $|c|$ changes would be 1-minimal if removing any single change would cause the failure to disappear. While removing two or more changes at once may result in an even smaller, still failing test case, every single change on its own is *significant in reproducing the failure.*

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

1-minimality is what we should be aiming at. However, given, say, a failure-inducing input of 100,000 lines, we cannot simply remove each individual line in order to minimize it. Thus, we need an effective algorithm to reduce our test case efficiently.

B. A Minimizing Algorithm

What do humans do in order to minimize test cases? One possibility: they use *binary search.* If $c_{\text{✘}}$ contains only one change, then $c_{\text{✘}}$ is minimal by definition. Otherwise, we *partition* $c_{\text{✘}}$ into two subsets Δ_1 and Δ_2 with similar size and test each of them. This gives us three possible outcomes:

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	.		

Fig. 4. Quick minimization of test cases

Reduce to Δ_1. The test of Δ_1 fails—Δ_1 is a smaller test case.[6]
Reduce to Δ_2. Testing Δ_1 does not fail, but testing Δ_2 fails—Δ_2 is a smaller test case.
Ignorance. Both tests pass, or are unresolved—neither Δ_1 nor Δ_2 qualify as possible simplifications.

In the first two cases, we can simply continue the search in the failing subset, as illustrated in Figure 4. Each line of the diagram shows a configuration. A number i stands for an included change δ_i; a dot stands for an excluded change. Change 7 is the minimal failing test case—and it is isolated in just a few steps.

Given sufficient knowledge about the nature of our input, we can certainly partition any test case into *two* subsets such that at least one of them fails the test. But what if this knowledge is insufficient, or not present at all?

Let us begin with the worst case: after splitting up $c_{\text{✘}}$ into subsets, all tests pass or are unresolved—ignorance is complete. All we know is that $c_{\text{✘}}$ as a whole is failing. How do we increase our chances of getting a failing subset?

- By testing *larger* subsets of $c_{\text{✘}}$, we increase the chances that the test fails—the difference from $c_{\text{✘}}$ is smaller. On the other hand, a smaller difference means a slower progression—the test case is not halved, but reduced by a smaller amount.
- By testing *smaller* subsets of $c_{\text{✘}}$, we get a faster progression in case the test fails. On the other hand, the chances that the test fails are smaller.

These specific methods can be combined by partitioning $c_{\text{✘}}$ into a *larger number of subsets* and testing each (small) Δ_i as well as its (large) complement $\nabla_i = c_{\text{✘}} - \Delta_i$—until each subset contains only one change, which gives us the best chance to get a failing test case. The disadvantage, of course, is that more subsets means more testing.

This is what can happen. Let n be the number of subsets $\Delta_1, \ldots, \Delta_n$. Testing each Δ_i and its complement $\nabla_i = c_{\text{✘}} - \Delta_i$, we have four possible outcomes (Figure 5):

Reduce to subset. If testing any Δ_i fails, then Δ_i is a smaller test case. Continue reducing Δ_i with $n = 2$ subsets.
This reduction rule results in a classical "divide and conquer" approach. If one can identify a smaller part of the test case that is failure-inducing on its own, then this rule helps in narrowing down the test case efficiently.
Reduce to complement. If testing any $\nabla_i = c_{\text{✘}} - \Delta_i$ fails, then ∇_i is a smaller test case. Continue reducing ∇_i with $n - 1$ subsets.

[5] To be precise, Corollary 7 tells us the results of *test*$(\emptyset)$ and *test*$(c_{\text{✘}})$, such that only $2^{|c_{\text{✘}}|} - 2$ subsets need to be tested, but this does not help much.

[6] Since Δ_1 and Δ_2 have similar size, there is no need for testing Δ_2 as well if testing Δ_1 already fails.

Minimizing Delta Debugging Algorithm

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

$$\mathit{ddmin}(c_{\text{✘}}) = \mathit{ddmin}_2(c_{\text{✘}}, 2) \quad \text{where}$$

$$\mathit{ddmin}_2(c'_{\text{✘}}, n) = \begin{cases} \mathit{ddmin}_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \dots, n\} \cdot \mathit{test}(\Delta_i) = \text{✘} \text{ (“reduce to subset”)} \\ \mathit{ddmin}_2\big(\nabla_i, \max(n-1, 2)\big) & \text{else if } \exists i \in \{1, \dots, n\} \cdot \mathit{test}(\nabla_i) = \text{✘} \text{ (“reduce to complement”)} \\ \mathit{ddmin}_2\big(c'_{\text{✘}}, \min(|c'_{\text{✘}}|, 2n)\big) & \text{else if } n < |c'_{\text{✘}}| \text{ (“increase granularity”)} \\ c'_{\text{✘}} & \text{otherwise (“done”).} \end{cases}$$

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

Fig. 5. Minimizing Delta Debugging algorithm

Step	Test case									*test*	
1	$\Delta_1 = \nabla_2$	1	2	3	4	.	.	.	.	?	Testing Δ_1, Δ_2
2	$\Delta_2 = \nabla_1$	.	.	.	.	5	6	7	8	?	$\Rightarrow$ Increase granularity
3	Δ_1	1	2	.	.	.	.	.	.	?	Testing $\Delta_1, \dots, \Delta_4$
4	Δ_2	.	.	3	4	.	.	.	.	✔	
5	Δ_3	.	.	.	.	5	6	.	.	✔	
6	Δ_4	.	.	.	.	.	.	7	8	?	
7	∇_1	.	.	3	4	5	6	7	8	?	Testing complements
8	∇_2	1	2	.	.	5	6	7	8	✘	$\Rightarrow$ Reduce to $c'_{\text{✘}} = \nabla_2$; continue with $n = 3$
9	Δ_1	1	2	.	.	.	.	.	.	?*	Testing $\Delta_1, \Delta_2, \Delta_3$
10	Δ_2	.	.	.	.	5	6	.	.	✔*	* same *test* carried out in an earlier step
11	Δ_3	.	.	.	.	.	.	7	8	?*	
12	∇_1	.	.	.	.	5	6	7	8	?	Testing complements
13	∇_2	1	2	.	.	.	.	7	8	✘	$\Rightarrow$ Reduce to $c'_{\text{✘}} = \nabla_2$; continue with $n = 2$
14	$\Delta_1 = \nabla_2$	1	2	.	.	.	.	.	.	?*	Testing Δ_1, Δ_2
15	$\Delta_2 = \nabla_1$	.	.	.	.	.	.	7	8	?*	$\Rightarrow$ Increase granularity
16	Δ_1	1	.	.	.	.	.	.	.	?	Testing $\Delta_1, \dots, \Delta_4$
17	Δ_2	.	2	.	.	.	.	.	.	✔	
18	Δ_3	.	.	.	.	.	.	7	.	?	
19	Δ_4	.	.	.	.	.	.	.	8	?	
20	∇_1	.	2	.	.	.	.	7	8	?	Testing complements
21	∇_2	1	.	.	.	.	.	7	8	✘	$\Rightarrow$ Reduce to $c'_{\text{✘}} = \nabla_2$; continue with $n = 3$
22	Δ_1	1	.	.	.	.	.	.	.	?*	Testing $\Delta_1, \dots, \Delta_3$
23	Δ_2	.	.	.	.	.	.	7	.	?*	
24	Δ_3	.	.	.	.	.	.	.	8	?*	
25	∇_1	.	.	.	.	.	.	7	8	?	Testing complements
26	∇_2	1	.	.	.	.	.	.	8	?	
27	∇_3	1	.	.	.	.	.	7	.	?	Done
Result		1	.	.	.	.	.	7	8		

Fig. 6. Minimizing a test case with increasing granularity

Why do we continue with $n - 1$ and not two subsets here? Because the granularity stays the same: Splitting ∇_i into $n-1$ subsets means that the subsets of ∇_i are identical to the subsets Δ_i of $c_{\text{✘}}$. Every subset of $c_{\text{✘}}$ eventually gets tested.

As an example, assume $n = 32$ and ∇_{30} fails. If we continue with $n = 31$, the recursive *ddmin* call splits ∇_{30} into $n = 31$ subsets. The subsets Δ_1 to Δ_{30} have already been tested before. If we realize the *test* function such that it keeps track of tests that already have been run, the next new test would be one of the complements ∇_i—we'd simply continue removing small chunks.

If we continued with two subsets instead, we would have to work our way down with $n = 2, 4, 8, \dots$ until the initial granularity of $n = 32$ is reached again.

Increase granularity. Otherwise (that is, no test failed), try again with $2n$ subsets. (Should $2n > |c_{\text{✘}}|$ hold, try again with

$|c_{\text{✘}}|$ subsets instead, each containing one change.) This results in at most twice as many tests, but increases chances for failure. *Done.* The process is repeated until granularity can no longer be increased (that is, the next n would be larger than $|c_{\text{✘}}|$). In this case, we have already tried removing every single change individually without further failures: the resulting change set is minimal.

As an example, consider Figure 6, where the minimal test case consists of the changes 1, 7, and 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 begin with partitioning the total set of changes in two halves—but none of them passes the test. We continue with granularity increased to 4 subsets (Step 3–6). When testing the complements, the set ∇_2 fails, thus removing changes 3 and 4. We continue with splitting ∇_2 into three subsets. The next three tests (Steps 9–11) have already been carried out and need not be repeated (marked with *). When testing ∇_2 (Step 13), changes 5 and 6 can be eliminated. We increase granularity to 4 subsets and test each (Steps 16–19), before the last complement ∇_2 (Step 21) eliminates change 2. Only changes 1, 7, and 8 remain; Steps 25–27 show that none of these changes can be eliminated. To minimize this test case, a total of 19 different tests was required.

C. Properties of ddmin

We close with some formal properties of *ddmin*. First, *ddmin* eventually returns a 1-minimal test case:

Proposition 11 (*ddmin* minimizes) *For any $c \subseteq c_{\text{✘}}$, ddmin(c) is 1-minimal in the sense of definition 10.* *Proof: According to the ddmin definition (Figure 5), ddmin$(c'_{\text{✘}})$ returns $c'_{\text{✘}}$ only if $n \geq |c'_{\text{✘}}|$ and test$(\nabla_i) \neq$ ✘ for all $\Delta_1, \ldots, \Delta_n$ where $\nabla_i = c'_{\text{✘}} - \Delta_i$. If $n \geq |c'_{\text{✘}}|$, then $|\Delta_i| = 1$ and $|\nabla_i| = |c| - 1$. Since all subsets of $c' \subset c'_{\text{✘}}$ with $|c'_{\text{✘}}| - |c'| = 1$ are in $\{\nabla_1, \ldots, \nabla_n\}$ and test$(\nabla_i) \neq$ ✘ for all ∇_i, the condition of definition 10 applies and c is 1-minimal.* ■

In the worst case, *ddmin* takes $|c_{\text{✘}}|^2 + 3|c_{\text{✘}}|$ tests:

Proposition 12 (*ddmin* complexity, worst case) *The number of tests carried out by ddmin$(c_{\text{✘}})$ is $|c_{\text{✘}}|^2 + 3|c_{\text{✘}}|$ in the worst case.* *Proof: The worst case can be divided into two phases: First, every test has an unresolved result until we have a maximum granularity of $n = |c_{\text{✘}}|$; then, testing only the last complement results in a failure until $n = 2$ holds.*

- *In the first phase, every test has an unresolved result. This results in a re-invocation of ddmin$_2$ with a doubled number of subsets, until $|\Delta_i| = 1$ holds. The number of tests to be carried out is $2+4+8+\cdots+2|c_{\text{✘}}| = 2|c_{\text{✘}}| + |c_{\text{✘}}| + \frac{|c_{\text{✘}}|}{2} + \frac{|c_{\text{✘}}|}{4} + \cdots = 4|c_{\text{✘}}|$.*
- *In the second phase, the worst case is that testing the* last *complement ∇_n fails; consequently, ddmin$_2$ is re-invoked with ddmin$_2(\nabla_n, |c_{\text{✘}}| - 1)$. This results in $|c_{\text{✘}}| - 1$ calls of ddmin, with two tests per call, or $2(|c_{\text{✘}}| - 1) + 2(|c_{\text{✘}}| - 2) + \cdots + 2 = 2 + 4 + 6 + \cdots + 2(|c_{\text{✘}}| - 1) = |c_{\text{✘}}|(|c_{\text{✘}}| - 1) = |c_{\text{✘}}|^2 - |c_{\text{✘}}|$ tests.*

The overall number of tests is thus $4|c_{\text{✘}}| + |c_{\text{✘}}|^2 - |c_{\text{✘}}| = |c_{\text{✘}}|^2 + 3|c_{\text{✘}}|$. ■

In practice, however, it is unlikely that an n-character input requires $n^2 + 3n$ tests. The "divide and conquer" rule of *ddmin* takes care of quickly narrowing down failure-inducing parts of the input:

Proposition 13 (*ddmin* complexity, best case) *If there is only one failure-inducing change $\Delta_i \in c_{\text{✘}}$, and all test cases that include Δ_i cause a failure as well, then the number of tests t is limited by $t \leq 2\log_2(|c_{\text{✘}}|)$.* *Proof: Under the given conditions, the test of either initial subset Δ_1 or Δ_2 will fail; $n = 2$ always holds. Thus, the overall complexity is that of a binary search.* ■

Whether this "best case" efficiency applies depends on our ability to break down the input into smaller chunks that result in determined (or better: failing) test outcomes. Consequently, the more knowledge about the structure of the input we have, the better we can identify possibly failure-inducing subsets, and the better is the overall performance of *ddmin*.

The surprising thing, however, is that even with *no knowledge about the input structure at all,* the *ddmin* algorithm has sufficient performance—at least in the case studies we have examined. This is illustrated in the following section.

IV. Case Studies

> *When you've cut away as much* HTML, CSS, *and JavaScript as you can, and cutting away any more causes the bug to disappear, you're done.*
>
> — *Mozilla BugAThon call*

Let us now turn to some real-life failures and simplify failure-inducing input. We discuss examples from the GNU C compiler, Mozilla, and various UNIX utilities subjected to random fuzz input.

A. GCC gets a Fatal Signal

The C program in Figure 7 not only demonstrates some particular nasty aspects of the language, it also causes the GNU C compiler (GCC) to crash—at least, when using version 2.95.2 on Intel-Linux with optimization enabled.

Before crashing, GCC grabs all available memory for its stack, such that other processes may run out of resources and die.[7] The latter can be prevented by limiting the stack memory available to GCC, but the effect remains:

```
$ (ulimit -H -s 256; gcc -O bug.c)
gcc: Internal compiler error:
     program cc1 got fatal signal 11
$ _
```

The GCC error message (and the resulting core dump) help GCC maintainers only; as ordinary users, we must now narrow down the failure-inducing input in `bug.c`—and *minimize* `bug.c` in order to file in a bug report.

[7] The author denies any liability for damage caused by repeating this experiment.

```
#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);
}
```

Fig. 7. The `bug.c` program that crashes GNU CC

In the case of GCC, the passing program run is the empty input. For the sake of simplicity, we modeled a *change* as the *insertion of a single character.* This means that

- $r_{\checkmark}$ is running GCC with an empty input
- $r_{\text{✘}}$ means running GCC with `bug.c`
- each change δ_i inserts the i-th character of `bug.c`
- partitioning $c_{\text{✘}}$ means partitioning the input into parts.

No special effort was made to exploit syntactic or semantic knowledge about C programs; consequently, we expected a large number of test cases to be invalid C programs.

To minimize `bug.c`, we implemented the *ddmin* algorithm of Figure 5 into our WYNOT prototype[8]. The *test* procedure would create the appropriate subset of `bug.c`, feed it to GCC, return ✘ iff GCC had crashed, and ✔ otherwise. The results of this WYNOT run are shown in Figure 8.

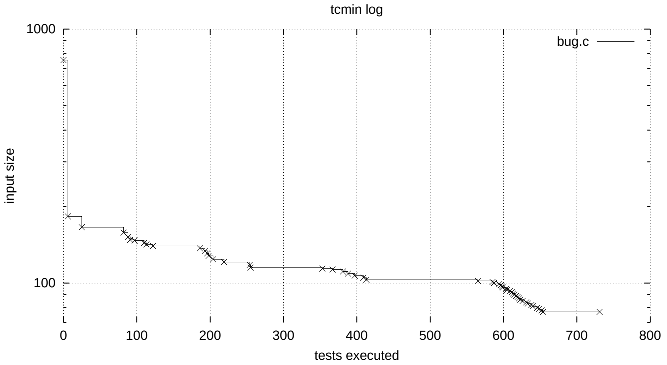


Fig. 8. Minimizing GCC input `bug.c`

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

TABLE I
GCC OPTIMIZATION OPTIONS

After the first two tests, WYNOT has already reduced the input size from 755 characters to 377 and 188 characters, respectively—the test case now only contains the *mult* function. Reducing *mult*, however, takes time: only after 731 more tests (and 34 seconds)[9] do we get a test case that can not be minimized any further. Only 77 characters are left:

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

This test case is 1-minimal—no single character can be removed without removing the failure. Even every single superfluous whitespace has been removed, and the function name has shrunk from *mult* to a single t. (At least, we now know that neither whitespace nor function name were failure-inducing!)

Figure 9 shows an excerpt of the Delta Debugging log: From "z[0]" to "return", we see how the *ddmin* algorithm tries to remove every single change (= character) in order to minimize the input even further—but each test results in a syntactically invalid program.

As GCC users, we can now file in the one-liner as a minimal bug report. But where in GCC does the failure actually occur? We already know that the failure is associated with optimization: If we remove the `-O` option to turn off optimization, the failure disappears. Could it be possible to keep optimization turned on, but to influence it in a way that the failure disappears?

The GCC documentation lists 31 options that can be used to influence optimization on Linux, shown in Table I. It turns out that applying *all of these options* causes the failure to disappear:

[8] WYNOT = "Worked Yesterday, NOt Today"

[9] All times were measured on a Linux PC with a 500 MHz Pentium III processor. The time given is the CPU user time of our WYNOT prototype as measured by the UNIX kernel; it includes all spawned child processes (such as the GCC run in this example).

```
714 t(double z[],int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*(z[0]+0);}return z[n];} ?
714 t(double z[],int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*(z[0]+0);}return z[n];} ?
715 t(double z[],int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*(z[0]+0);}return z[n];} ?
716 t(double z[],int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*(z[0]+0);}return z[n];} ?
717 t(double z[],int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*(z[0]+0);}return z[n];} ?
718 t(double z[],int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*(z[0]+0);}return z[n];} ?
719 t(double z[],int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*(z[0]+0);}return z[n];} ?
720 t(double z[],int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*(z[0]+0);}return z[n];} ?
721 t(double z[],int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*(z[0]+0);}return z[n];} ?
722 t(double z[],int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*(z[0]+0);}return z[n];} ?
 :                                      :
 :                                      :                                        ?
733 t(double z[],int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*(z[0]+0);}return z[n];} ✘
```

Fig. 9. Minimizing GCC input `bug.c`

```
$ gcc -O -ffloat-store -fno-default-inline \
    -fno-defer-pop ...-fstrict-aliasing bug.c
$ _
```

This means that some option(s) in the list *prevent* the failure. We can use test case minimization in order to find the preventing option(s). This time, each δ_i stands for *removing* a GCC option from Table I: Having all δ_i applied means to run GCC with no option (failing), and having no δ_i applied means to run GCC with all options (passing).

This WYNOT run is a straight-forward "divide and conquer" search, shown in Figure 10. After 7 tests (and less than a second), the single option *–ffast-math* is found which prevents the failure:

```
$ gcc -O -ffast-math bug.c
$ _
```

Unfortunately, the *–ffast-math* option is a bad candidate for working around the failure, because it may alter the semantics of the program. We remove *–ffast-math* from the list of options and make another WYNOT run. Again after 7 tests, it turns out the option *–fforce-addr* also prevents the failure:

```
$ gcc -O -fforce-addr bug.c
$ _
```

Are there any other options that prevent the failure? Running GCC with the remaining 29 options shows that the failure is still there; so it seems we have identified all failure-preventing options. And this is what we can send to the GCC maintainers:

1. The minimal test case
2. "The failure occurs only with optimization."
3. "*–ffast-math* and *–fforce-addr* prevent the failure."

Still, we cannot identify a place in the GCC code that causes the problem. On the other hand, we have identified as many *failure circumstances* as we can. In practice, program maintainers can easily enhance their automated regression test suites such that the failure circumstances are automatically simplified for any failing test case.

B. Mozilla Cannot Print

As a further case study, we wanted to simplify a real-world Mozilla test case and thus contribute to the Mozilla BugAThon. A search in Bugzilla, the Mozilla bug database, shows us bug #24735, reported by *anantk@yahoo.com*:

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

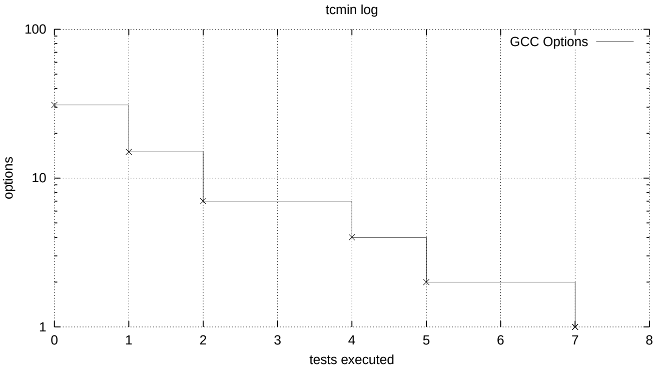


Fig. 10. Minimizing GCC options

In this case, the Mozilla input consists of two items: The *sequence of input events*—that is, the succession of mouse motions, pressed keys, and clicked buttons—and the HTML *code* of the erroneous WWW page. We used the XLAB *capture/replay* tool [5] to run Mozilla while capturing all user actions and logging them to a file. We could easily reproduce the error, creating an XLAB log with 711 recorded X events. Our WYNOT tool would now use XLAB to *replay* the log and feed Mozilla with the recorded user actions, thus automating Mozilla execution.

In a first run, we wanted to know whether all actions in the bug report were actually necessary. We thus subjected the log to test case minimization, in order to find a *failure-inducing minimum of user actions.* Out of the 711 X events, only 95 were induced by user actions—that is, moving the mouse pointer, pressing or releasing the mouse button, and pressing or releasing a key on the keyboard. (The other events were induced by the X server, such as notifications that the window should be redrawn.) These 95 user actions could easily be filtered out automatically by event type and were then subjected to minimization.

The *test* function would start Mozilla and use XLAB to replay the given set of user actions and then wait for a few seconds. If Mozilla crashed during this interval, *test* would return ✘; other-

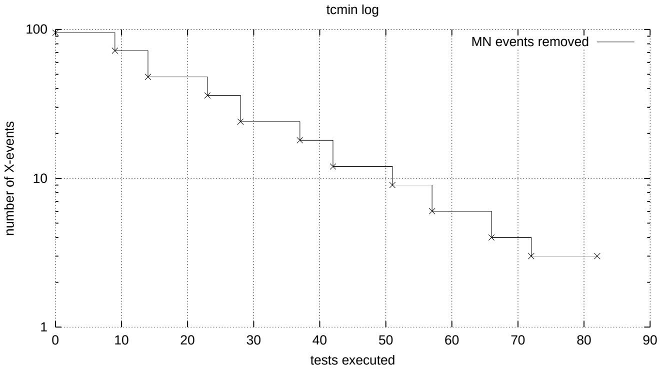


Fig. 11. Minimizing Mozilla user actions

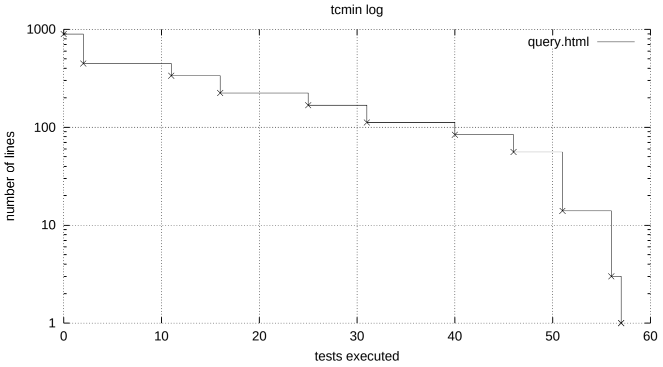


Fig. 12. Minimizing Mozilla HTML input

wise, *test* would terminate Mozilla and return ✔.[10]

The results of this run are shown in Figure 11. After 82 test runs (or 21 minutes), only 3 out of 95 user actions are left:

1. Press the *P* key while the *Alt* modifier key is held. (Invoke the *Print* dialog.)
2. Press *mouse button 1* on the *Print* button without a modifier. (Arm the *Print* button.)
3. Release *mouse button 1*. (Start printing.)

User actions removed include moving the mouse pointer, selecting the *Print to file* option, altering the default file name, setting the print margins to *.50*, and releasing the *P* key before clicking on *Print*—all this is irrelevant in producing the failure.[11]

Since the user actions can hardly be further generalized, we turn our attention to another input source–the failure-inducing HTML code. The original *Search for bug* page has a length of 39094 characters or 896 lines; an excerpt is shown in Figure 1. In order to minimize the HTML code, we chose a *hierarchical* approach: In a first run, we wanted to minimize the *number of lines* (that is, each Δ_i was identified with a line); in a later run, we wanted to minimize the failure-inducing line(s) according to single characters.

The results of the *lines* run are shown in Figure 12. After 57 test runs, the *ddmin* algorithm minimizes the original 896 lines to a 1-line input:

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

This is the HTML input which causes Mozilla to crash when being printed. As in the GCC example of Section IV-A, the actual failure-inducing input is very small. It should be noted, though, that the original HTML code contains multiple `SELECT` tags; Delta Debugging returns only one of them.[12] Further minimization by characters, as shown in Figure 2, reveals that the attributes of the `SELECT` tag are not relevant for reproducing the failure, either—the single input

```
<SELECT>
```

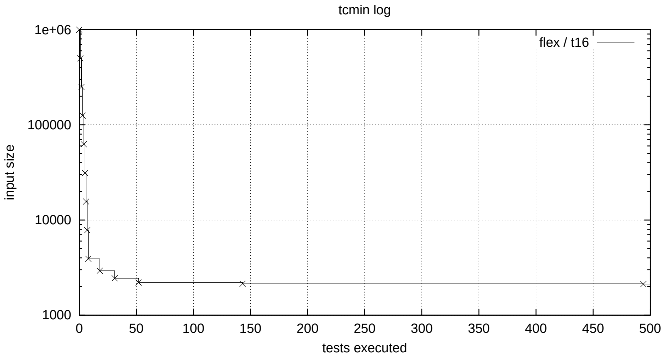


Fig. 13. Minimizing FLEX fuzz input

already suffices for reproducing the failure. Overall, we obtain the following self-contained minimized bug report:

→ Create an HTML page containing "`<SELECT>`"
→ Load the page and print it using *Alt+P* and *Print*.
→ The browser crashes with a segmentation fault.

or even simpler:

→ Printing "`<SELECT>`" causes a crash.

In principle, this minimization procedure could easily be applied automatically on the 12,479 open bugs listed in the Bugzilla database[13]—provided that the bug reports can be reproduced automatically. All one needs is an HTML input, a sequence of user actions, an observable failure—and a little time to let the computer simplify the failure-inducing input.

C. Minimizing Fuzz

In a classical experiment [6], [7], Bart Miller and his team examined the robustness of UNIX utilities and services by sending them *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 when being fed with fuzz input.

We wanted to know how well the *ddmin* algorithm performs in minimizing the fuzz input sequences. We examined a subset of the UNIX utilities listed in Miller's paper: NROFF (format documents for display), TROFF (format documents for typesetter), FLEX (fast lexical analyzer generator), CRTPLOT (graphics filter

[10] As in all testing, it is always a good idea to set an upper time bound for test cases.

[11] It is relevant, though, that the mouse button be pressed before it is released.

[12] If desired, one could easily re-invoke Delta Debugging on the remainder to search for other independent failure causes. In practice, though, we expect that after Delta Debugging has simplified a test case, first the error is fixed. Then, the test is repeated with the fixed program. If the failure persists, then Delta Debugging can find the next failure cause.

[13] as of 15 February 2001, 13:00 GMT

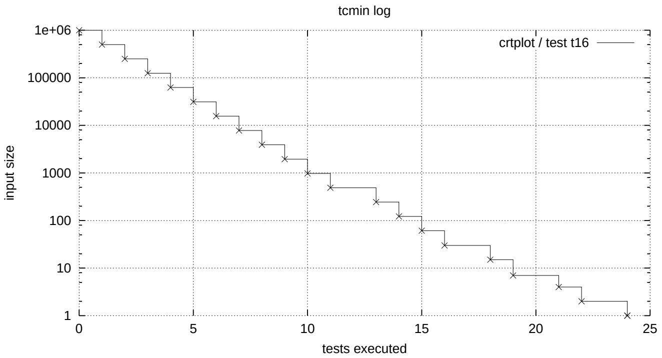


Fig. 14. Minimizing CRTPLOT fuzz input

for various plotters), UL (underlining filter), and UNITS (convert quantities).

We set up 16 different fuzz inputs, differing in size (10^3 to 10^6 characters) and content (whether all characters or only printable characters were included, and whether NUL characters were included or not). As shown in Table II(b), Miller's results still apply—at least on Sun's Solaris 2.6 operating system: out of $6 \times 16 = 96$ test runs, the utilities crashed 42 times (43%).

We applied our WYNOT tool in all 42 cases to minimize the failure-inducing fuzz input. In a first series, our *test* function would simply return ✘ if the input made the program crash, and ✔, otherwise. Table II(c) shows the resulting input sizes; Table II(d) lists the number of tests required. Depending on the crash cause, the programs could be partitioned into two groups:

- The first group of programs shows obvious *buffer overrun* problems.
  - FLEX, the most robust utility, crashes on sequences of 2,121 or more non-newline and non-NUL characters (t_{14}–t_{15}).
  - UL crashes on sequences of 516 or more printable non-newline characters (t_5–t_8, t_{13}–t_{16}).
  - UNITS crashes on sequences of 77 or more 8-bit characters (t_2–t_4 and t_{11}–t_{12}).

Figure 13 shows the first 500 tests of the WYNOT run for FLEX and t_{16}. After 494 tests, the remaining size of 2,122 characters is already close to the final size; however, it takes more than 10,000 further tests to eliminate one more character.

- The second group of programs appears vulnerable to *random commands*.
  - NROFF and TROFF crash
    - on *malformed commands* like `"\\D^J%0F"`[14] (NROFF, t_6), and
    - on *8-bit input* such as `"\302\n"` (TROFF, t_1)
  - CRTPLOT crashes on the one-letter inputs `"t"` (t_1) and `"f"` (t_5, t_9, t_{13}–t_{16}).

The WYNOT run for CRTPLOT and t_{16} is shown in Figure 14. It takes 24 tests to minimize the fuzz input of 10^6 characters to the single failure-inducing character.

Again, all test runs can be (and have been) entirely automated. This allows for *massive automated stochastic testing*, where programs are fed with fuzz input in order to reveal defects. As soon as a failure is detected, input minimization can generalize the large fuzz input to a minimal bug report.

D. The Precision Effect

In the fuzz examples from Section IV-C, our *test* function would return ✘ whenever a program crashed—regardless of further circumstances. This ignorance may lead to a problem: the minimized input may cause a *different failure* than the original test case.

In the fuzz examples, a different failure may be tolerable: Just as in the Mozilla case study (Section IV-B), there may be multiple independent failure causes, and eventually, we must fix them all. In the context of debugging, though, it is important that the causes for the *original failure* be isolated.

As a consequence, we repeated our test runs with an *increased precision,* which would also compare the location of the failure. As location, we used the *backtrace*—that is, the current program counter and the stack of calling functions at the moment of the crash; the values of arguments and local variables were not part of the backtrace, though.

- The *test* function would return ✘ only if the program crashed and if the backtrace of the failure was identical to the original backtrace.
- If the program failed, but with a different backtrace, *test* would return **?**.
- If the program did not crash, *test* would return ✔.

As shown in Table II(e), this increase in precision resulted in larger minimized test cases for NROFF, TROFF, and FLEX; the other three programs are unchanged. As an example, the NROFF input t_1 has been minimized from 10^3 to 60 characters; with lower precision (Table II(c)), only 2 characters were left. This indicates that the 2 characters from Table II(c) induce a failure different from the original one; only the 60 characters in Table II(e) induce the same backtrace.

Besides the backtrace, there is more one could compare: the entire memory contents, for instance, or the full execution traces. One will find, though, that higher precision will always increase the size of the minimized test case. This is so because only the complete original input can induce the complete original failure; and a complete comparison of behavior will make all of the original input significant (except for those parts, of course, which do not have any impact on the final program state at all). In practice, a simple backtrace as in our setting should provide sufficient precision.

V. ISOLATING FAILURE-INDUCING DIFFERENCES

> *So assess them to find out their plans,*
> *both the successful ones and the failures.*
> *Incite them to action in order to find out*
> *the patterns of movement and rest.*
>
> — Sun Tzu, *The Art of War*

The case studies as discussed in Sections IV-C and IV-D exhibit a major weakness of the *ddmin* algorithm: The larger the size of the simplified input, the higher is the number of tests required. This is pretty obvious, because determining the 1-minimality of a test case with *n* entities requires at least *n* tests—each entity is individually removed and tested. Consequently,

[14] All input is shown in C string notation.

(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

(c) Size $|c'_✘|$ 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	–	–	–	–

(e) Size $|c'_✘|$ 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	–	–	–	–

TABLE II

MINIMIZING FAILURE-INDUCING FUZZ INPUT

with the simplified FLEX input of 2121 characters, the number of tests (Table II(d)) varies between 11589 (fuzz input of 10^4 characters) and 17960 (10^5 fuzz input characters); with high precision, the number of tests is between 34450 and 37454 (Table II(f)).

36,000 tests are not much of an issue if each individual test is fast. If a single test takes about 0.1 seconds, as in the FLEX case, the entire simplification requires 1 hour. However, if the tests are less trivial, or if the size of the simplified input is larger, we have a serious problem.

There are many pragmatic approaches to resolve this issue, such as stopping simplification as soon as a time limit is reached or as soon as the original test case is reduced by a certain amount. However, there is a better strategy. Rather than only cutting away while the failure persists, one can also *add differences* while the program still passes the test. To get the best efficiency, one can combine both approaches and *narrow down the set of differences* whenever a test either passes or fails.

A. Isolation Illustrated

This idea of *isolating the failure-inducing differences* is best illustrated in comparison to the "simplification" approach discussed so far. Figure 2 shows how *ddmin* simplifies the failure-inducing HTML line presented in Section IV-B: After 26 steps, the line is reduced to the single `<SELECT>` tag.

Figure 3 shows the alternative "isolation" approach. Again, as in *ddmin*, each time a test case fails, the smaller test case is used as a new failing test case. This minimizes the failing test case as well as the difference between the failing test case and the (initially empty) passing test case. However, each time a test case *passes*, the larger test case is used as *new passing test case*, thus minimizing the difference as well.

Before going into details of the algorithm, let us look at the results: After seven tests, the failure-inducing difference is narrowed down to one `<` character. Prefixing the passing test

```
SELECT␣NAty"␣MULTIPLE␣SIZE=7>
```

with a `<` character changes the `SELECT` text to an HTML `<SELECT>` tag, causing the failure when being printed. This example demonstrates the basic difference between simplification and isolation:

- *Simplification* means to make each part of the simplified test case relevant: removing any part makes the failure go away.
- *Isolation* means to find one relevant part of the test case: removing this particular part makes the failure go away.

In general, isolation is much more efficient than simplification. If we have a large failure-inducing input, isolating the difference will pinpoint a failure cause much faster than minimizing the test case—in Figure 3, isolating requires only 7 tests, while minimizing (Figure 2) required 26 tests.

On the other hand, focusing on the difference requires the programmer to keep the *common context* of both test cases in mind—that is, the passing test case. This imposes an extra load on the programmer. In the future, though, we might have debugging tools that highlight differences and commonalities between multiple runs. For such tools, having two test cases with a minimal difference is far preferable to having only one simplified test case. Already today, if the isolated difference consists of, say, two items to keep in mind, while the minimized test case consists of, say, a hundred items to keep in mind, the isolated difference may lead much faster to the fix.

Another important point is that the running time of the program is frequently proportional to the size of its input. In some cases, simplification may require a larger amount of tests, but a lower total running time, due to the smaller input. Consequently, even when isolating differences, we should take care to prefer simple test cases. In practice, intended use and available resources may result in a mix of both simplification and isolation.

B. An Isolating Algorithm

Let us now formally define the algorithm that isolates failure-inducing differences. How can we extend the *ddmin* algorithm to obtain the behavior as sketched in Figure 3? Our goal is to find two sets $c'_{\text{✔}}$ and $c'_{\text{✘}}$ such that $\emptyset = c_{\text{✔}} \subseteq c'_{\text{✔}} \subset c'_{\text{✘}} \subseteq c_{\text{✘}}$ holds and the difference $\Delta = c'_{\text{✘}} - c'_{\text{✔}}$ is minimal.

Again, we need to specify what we mean by minimality, now applied to differences instead of test cases. The definition of minimality follows Definition 9:

Definition 14 (Minimal failure-inducing difference) *Let $c'_{\text{✔}}$ and $c'_{\text{✘}}$ be two test cases with $\emptyset = c_{\text{✔}} \subseteq c'_{\text{✔}} \subset c'_{\text{✘}} \subseteq c_{\text{✘}}$. Their difference $\Delta = c'_{\text{✘}} - c'_{\text{✔}}$ is* minimal *if*

$$\forall \Delta_i \subset \Delta \cdot \mathit{test}(c'_{\text{✔}} \cup \Delta_i) \neq \text{✔} \wedge \mathit{test}(c'_{\text{✘}} - \Delta_i) \neq \text{✘}$$

holds.

Again, the number of subsets of Δ is exponential, so we resort to the same pragmatic approximation as in Definition 10:

Definition 15 (n-minimal difference) *Let $c'_{\text{✔}}$ and $c'_{\text{✘}}$ be defined as in Definition 14. Their difference $\Delta = c'_{\text{✘}} - c'_{\text{✔}}$ is n-*minimal *if*

$$\forall \Delta_i \subset \Delta \cdot |\Delta_i| \leq n \Rightarrow \left(\mathit{test}(c'_{\text{✔}} \cup \Delta_i) \neq \text{✔} \wedge \mathit{test}(c'_{\text{✘}} - \Delta_i) \neq \text{✘}\right)$$

holds. Consequently, Δ is 1-minimal *if*

$$\forall \delta_i \in \Delta \cdot \mathit{test}\big(c'_{\text{✔}} \cup \{\delta_i\}\big) \neq \text{✔} \wedge \mathit{test}\big(c'_{\text{✘}} - \{\delta_i\}\big) \neq \text{✘}$$

holds.

This is what we are aiming at: *to isolate a 1-minimal difference* between a passing and a failing test case.

It turns out that the original *ddmin* algorithm, as discussed in Section III-B can easily be extended to compute a 1-minimal difference rather than a minimal test case. Besides reducing the failing test case $c'_{\text{✘}}$ whenever a test fails, we now also *increase* the passing test case $c'_{\text{✔}}$ whenever a test passes. At all times, $c'_{\text{✔}}$ and $c'_{\text{✘}}$ act as lower and upper bound of the search space, which is systematically narrowed—like in a branch-and-bound algorithm, except that there is no backtracking.

This is what we have to do to extend *ddmin*:

1. Extend *ddmin* such that it works on two sets at a time:
   - The passing test case $c'_{\text{✔}}$ which is to be maximized (initially, $c'_{\text{✔}} = c_{\text{✔}} = \emptyset$ holds) and

General Delta Debugging Algorithm

Let *test* and $c_{\text{✘}}$ be given such that $test(\emptyset) = \text{✔} \wedge test(c_{\text{✘}}) = \text{✘}$ hold.
The goal is to find $(c'_{\text{✔}}, c'_{\text{✘}}) = dd(c_{\text{✘}})$ such that $\emptyset = c_{\text{✔}} \subseteq c'_{\text{✔}} \subset c'_{\text{✘}} \subseteq c_{\text{✘}}$, $test(c'_{\text{✔}}) = \text{✔}$, $test(c'_{\text{✘}}) = \text{✘}$, and $\Delta = c'_{\text{✘}} - c'_{\text{✔}}$ is 1-minimal.
The *general Delta Debugging algorithm* $dd(c_{\text{✘}})$ is

$$dd(c_{\text{✘}}) = dd_2(\emptyset, c_{\text{✘}}, 2) \quad \text{where}$$

$$dd_2(c'_{\text{✔}}, c'_{\text{✘}}, n) = \begin{cases} dd_2(c'_{\text{✔}}, c'_{\text{✔}} \cup \Delta_i, 2) & \text{if } \exists i \in \{1, \dots, n\} \cdot test(c'_{\text{✔}} \cup \Delta_i) = \text{✘ (“reduce to subset”)} \\ dd_2(c'_{\text{✘}} - \Delta_i, c'_{\text{✘}}, 2) & \text{else if } \exists i \in \{1, \dots, n\} \cdot test(c'_{\text{✘}} - \Delta_i) = \text{✔ (“increase to complement”)} \\ dd_2\big(c'_{\text{✔}} \cup \Delta_i, c'_{\text{✘}}, \max(n-1, 2)\big) & \text{else if } \exists i \in \{1, \dots, n\} \cdot test(c'_{\text{✔}} \cup \Delta_i) = \text{✔ (“increase to subset”)} \\ dd_2\big(c'_{\text{✔}}, c'_{\text{✘}} - \Delta_i, \max(n-1, 2)\big) & \text{else if } \exists i \in \{1, \dots, n\} \cdot test(c'_{\text{✘}} - \Delta_i) = \text{✘ (“reduce to complement”)} \\ dd_2\big(c'_{\text{✔}}, c'_{\text{✘}}, \min(2n, |\Delta|)\big) & \text{else if } n < |\Delta| \text{ (“increase granularity”)} \\ (c'_{\text{✔}}, c'_{\text{✘}}) & \text{otherwise (“done”)} \end{cases}$$

where $\Delta = c'_{\text{✘}} - c'_{\text{✔}} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |\Delta|/n$ holds.
The recursion invariant (and thus precondition) for dd_2 is $test(c'_{\text{✘}}) = \text{✘} \wedge test(c'_{\text{✔}}) = \text{✔} \wedge n \leq |\Delta|$.

Fig. 15. General Delta Debugging algorithm

- The failing test case $c'_{\text{✘}}$ which is to be minimized (initially, $c'_{\text{✘}} = c_{\text{✘}}$ holds).

2. Compute subsets Δ_i as subsets of $\Delta = c'_{\text{✘}} - c'_{\text{✔}}$ (instead of subsets of $c'_{\text{✘}}$)
3. Change the rule “Reduce to subset” such that $c'_{\text{✔}} \cup \Delta_i$ is tested (and passed to the recursive call) instead of Δ_i.
4. Introduce two additional rules for passing test cases:

Increase to complement. If $c'_{\text{✘}} - \Delta_i$ passes for any subset Δ_i, then $c'_{\text{✘}} - \Delta_i$ is a larger passing test case. Continue reducing the difference between $c'_{\text{✘}} - \Delta_i$ and $c'_{\text{✘}}$.
This is just the complement of the “reduce to subset” rule in *ddmin*.
Increase to subset. If $c'_{\text{✔}} \cup \Delta_i$ passes for any subset Δ_i, then $c'_{\text{✔}} \cup \Delta_i$ is a larger passing test case.
Again, this is just the complement of the “reduce to complement” rule in *ddmin*.
As a consequence of the additional rules, the “increase granularity” rule only applies if all previous tests turn out unresolved.

The full *dd* algorithm is shown in Figure 15.

C. Properties of dd

Being based on *ddmin*, the *dd* algorithm inherits most properties. In particular, *dd* returns a 1-minimal difference and has the same worst-case number of tests:

Proposition 16 (*dd* minimizes) *For any $c \subseteq c_{\text{✘}}$, let $(c'_{\text{✔}}, c'_{\text{✘}}) = dd(c)$. Then, $\Delta = c'_{\text{✘}} - c'_{\text{✔}}$ is 1-minimal in the sense of definition 15. Proof: (Compare proof of proposition 11) According to the dd definition (Figure 15), $dd_2(c'_{\text{✔}}, c'_{\text{✘}}, n)$ returns $(c'_{\text{✔}}, c'_{\text{✘}})$ only if $n \geq |\Delta|$ where $\Delta = c'_{\text{✘}} - c'_{\text{✔}} = \Delta_1 \cup \cdots \cup \Delta_n$; that is, $|\Delta_i| = 1$ and $\Delta_i = \{\delta_i\}$ hold for all i.*

Furthermore, for dd_2 to return $(c'_{\text{✔}}, c'_{\text{✘}})$, the conditions $test(c'_{\text{✔}} \cup \Delta_i) \neq \text{✘}$, $test(c'_{\text{✘}} - \Delta_i) \neq \text{✔}$, $test(c'_{\text{✔}} \cup \Delta_i) \neq \text{✔}$, and $test(c'_{\text{✘}} - \Delta_i) \neq \text{✘}$ must hold. These are the conditions of definition 15; consequently, Δ is 1-minimal. ■

Proposition 17 (*dd* complexity, worst case) *The number of tests carried out by $dd(c_{\text{✘}})$ is $|c_{\text{✘}}|^2 + 3|c_{\text{✘}}|$ in the worst case. Proof: The worst case is the same as in Proposition 12; hence, the number of tests is the same.* ■

Actually, *ddmin* is an instance of *dd*: if *test* returns ✔ only for $c_{\text{✔}}$: in this case, $c'_{\text{✔}} = c_{\text{✔}} = \emptyset$ always holds and only $c'_{\text{✘}}$ is minimized.[15] However, *dd* is much more efficient than *ddmin* if there are no unresolved test cases; this “best case” even requires half as many tests as *ddmin*.

Proposition 18 (*dd* complexity, best case) *If all tests return either ✔ or ✘, then the number of tests t is limited by $t \leq \log_2(|c_{\text{✘}}|)$. Proof: We decompose $\Delta = \Delta_1 \cup \Delta_2 = c'_{\text{✘}} - c'_{\text{✔}}$. Under the given conditions, the test of $c'_{\text{✔}} \cup \Delta_1 = c'_{\text{✘}} - \Delta_2$ will either pass or fail; $n = 2$ always holds. This is equivalent to a classical binary search algorithm over a sorted array: with each recursion, the difference is reduced by 1/2; the overall complexity is the same.* ■

Proposition 18 tells us what makes the search for the `SELECT` tag so efficient: There were no unresolved test outcomes in the Mozilla test case. In fact, when there are no unresolved test outcomes, *dd* always returns a single failure-inducing change:

Corollary 19 (Size of failure-inducing difference, best case) *If all tests return either ✔ or ✘, then $|dd(c_{\text{✘}})| = 1$ holds. Proof: Follows directly from the equivalence to binary search, as shown in Proposition 18.* ■

However, these “best cases” need not always be given—the more unresolved test outcomes we have, the more tests will be required. Let us see how *dd* behaves in practice when there are unresolved test outcomes.

[15] There is another instance of *dd*, which might be called a “maximizing” algorithm; it minimizes the difference only by extending the passing test case. This *ddmax* variant is obtained if *test* returns ✘ only for $c_{\text{✘}}$: then, $c'_{\text{✘}} = c_{\text{✘}}$ always holds and $c'_{\text{✔}}$ is maximized.

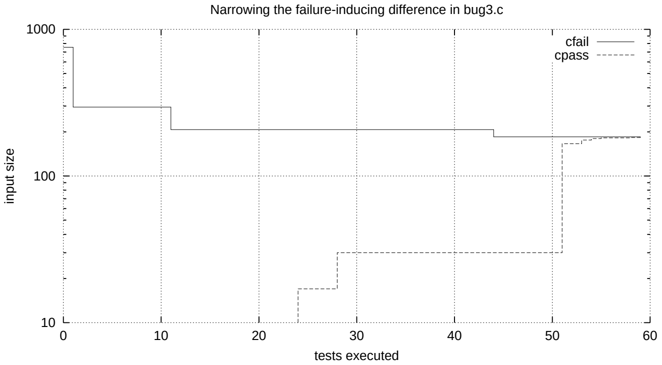


Fig. 16. Narrowing down the failure-inducing difference

(a) failing 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];
}
```

(b) passing program

```
#define SIZE 20

double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }

  return z[n];
}
```

Fig. 17. A failure-inducing difference

VI. Case Studies Revisited

> *– How do they know the load limit on bridges, Dad?*
> *– They drive bigger and bigger trucks over the bridge until it breaks. Then they weigh the last truck and rebuild the bridge.*
>
> — Bill Watterson, *Calvin and Hobbes*

To demonstrate the difference in performance between *dd* and *ddmin*, we have repeated the GCC and fuzz case studies with the *dd* algorithm.

A. Isolating GCC Input

As a first example, reconsider the GCC example from Section IV-A. Since we are not interested in programs with invalid syntax, we set up the *test* function such that it would return ✔ if the compilation succeeded, ✘ if the compiler crashed, and **?** in all other cases (notably if the compilation failed).

With *ddmin*, it took us 731 tests to minimize the entire program. Isolating the difference requires but 59 tests (Figure 16), but nonetheless pinpoints to a relevant difference of 2 characters. As shown in Figure 17 it suffices to remove the assignment to i in the `mult` function to make the program work (Figure 17(b)). This suggests a problem with inlining the expression $i + j + 1$ in the array accesses $z[i]$ on the following line.

B. Isolating Fuzz Input

In a second example, we have repeated the high-precision fuzz experiments of Section IV-D with the *dd* algorithm—that is, the test outcome was **?** if the failure backtrace was different from the original backtrace.[16]

As shown in Table III(b), the number of test runs is much smaller for *dd* than for *ddmin*. Except for NROFF, the minimal failure-inducing difference is always just 1 character. Only NROFF, TROFF, and FLEX have any unresolved test outcomes (Table III(d)); for all others, the number of test runs (Table III(c)) is logarithmic in proportion to the input size as predicted in Proposition 18.

Table III(e) shows the size of the common context—that is, the size of the maximized passing input $c'_{\checkmark}$. In the UL example, for instance, we can see that adding one more character to the 515 passing ones causes the failure. Likewise, the FLEX buffer is overrun after adding one more character to a base of 7804 to 7811 characters. In all cases, the number of tests is significantly lower than with the *ddmin* algorithm.

VII. Related Work

> *When you have two competing theories which make exactly the same predictions, the one that is simpler is the better.*
>
> — *Occam's Razor*

As stated in the introduction, we are unaware of any other technique that would automatically simplify test cases to determine failure-inducing input. One important exception is the simplification of test cases which have been *artificially produced.* In [8], Don Slutz describes how to stress-test databases with generated SQL statements. After a failure has been produced, the test cases had to be simplified—after all, a failing 1,000-line SQL statement would not be taken seriously by the database vendor, but a 3-line statement would. This simplification was realized simply by undoing the earlier production steps and testing whether the failure still occurred.

In general, Delta Debugging determines circumstances that are relevant for producing a failure (in our case, parts of the program input). Such work has been conducted before. However, the previous work was always specific to a particular domain, and always only as simple binary search for a single circumstance. An example for such work is detecting a single failure-inducing component in an optimizing compiler [9].

The *dd* algorithm presented in this paper is a successor to the dd^+ algorithm presented in [3]. Like *dd*, dd^+ takes a set of changes and minimizes it according to a given test; in [3], these changes affected the program code and were obtained by comparing two program versions.

The main differences between *dd* and dd^+ are:

- dd^+ is not well-suited for failures induced by a large combination of changes. In particular, dd^+ does not guarantee a 1-minimal subset, which is why it is not suited for minimizing test cases.
- dd^+ assumes *monotonicity:* that is, whenever $test(c) = ✔$ holds, then $test(c') = ✔$ holds for every subset $c' \subseteq c$ as well. This assumption, which was found to be useful for changes to

[16] We also repeated the low-precision experiments. But since the test outcome was always ✔ or ✘, the experiments outcome just confirmed the predictions of Proposition 18 and Corollary 19.

(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) Size $|c'_{\text{✘}}| - |c'_{\text{✔}}|$ of minimized difference—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	1	6	1	1	–	1	1	1	2	1	17	1	–	–	–	–
TROFF	–	1	1	1	–	1	1	1	–	–	1	1	–	–	–	–
FLEX	–	–	–	–	–	–	–	–	–	–	–	–	–	1	1	1
CRTPLOT	1	–	–	–	1	–	–	–	1	–	–	–	1	1	1	1
UL	–	–	–	–	1	1	1	1	–	–	–	–	1	1	1	1
UNITS	–	1	1	1	–	–	–	–	–	–	1	1	–	–	–	–

(c) 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	24	84	23	30	–	15	19	21	161	62	473	24	–	–	–	–
TROFF	–	19	20	24	–	15	19	21	–	–	20	23	–	–	–	–
FLEX	–	–	–	–	–	–	–	–	–	–	–	–	–	23	51	33
CRTPLOT	12	–	–	–	12	–	–	–	12	–	–	–	12	15	20	22
UL	–	–	–	–	12	15	18	22	–	–	–	–	12	16	19	22
UNITS	–	15	19	22	–	–	–	–	–	–	19	22	–	–	–	–

(d) Number of unresolved test outcomes—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	9	62	3	6	–	0	0	0	119	38	390	2	–	–	–	–
TROFF	–	3	1	2	–	0	0	0	–	–	1	1	–	–	–	–
FLEX	–	–	–	–	–	–	–	–	–	–	–	–	–	10	29	15
CRTPLOT	0	–	–	–	0	–	–	–	0	–	–	–	0	0	0	0
UL	–	–	–	–	0	0	0	0	–	–	–	–	0	0	0	0
UNITS	–	0	0	0	–	–	–	–	–	–	0	0	–	–	–	–

(e) Size $|c'_{\text{✔}}|$ of common context—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	224	196	187	180	–	2105	2105	2105	147	259	227	145	–	–	–	–
TROFF	–	4921	3901	3877	–	2105	2105	2105	–	–	38211	38992	–	–	–	–
FLEX	–	–	–	–	–	–	–	–	–	–	–	–	–	7804	7808	7811
CRTPLOT	789	–	–	–	760	–	–	–	721	–	–	–	263	263	68	263
UL	–	–	–	–	523	523	523	523	–	–	–	–	515	515	515	515
UNITS	–	284	284	284	–	–	–	–	–	–	2731	2731	–	–	–	–

TABLE III

ISOLATING FAILURE-INDUCING DIFFERENCES IN FUZZ INPUT

program code, gave dd^+ a better performance when most tests produced determinate results.

We recommend *dd* as a general replacement for dd^+. To exploit monotonicity in *dd*, one can make $test(c)$ return ✔ whenever a superset of c has already passed the test, and ✘ whenever a subset of c has already failed the test.

VIII. FUTURE WORK

If you get all the way up to the group-signed T-Shirt, you can qualify for a stuffed animal as well by doing 12 more.

— *Mozilla BugAThon call*

Our future work will concentrate on the following topics:

Domain-specific simplification methods. Knowledge about the input structure can very much enhance the performance of the Delta Debugging algorithms. For instance, valid program inputs are frequently described by *grammars*; it would be nice to rely on such grammars in order to exclude syntactically invalid input right from the start. Also, with a formal input description, one could replace input by smaller *alternate input* rather than simply cutting it away. In the GCC example, one could try to replace arithmetic expressions by constants, or program blocks by no-ops; HTML input could be reduced according to HTML structure rules. Besides grammars, changes may also be constrained by explicit change constraints, as established in version control [10].

Optimization. In general, the abstract description of the Delta Debugging algorithms leaves a lot of flexibility in the actual implementation and thus provides “hooks” for several domain-specific optimizations:

- The implementation can choose how to *partition* the difference Δ into subsets Δ_i. This is the place where knowledge about the structure of the input comes in handy.
- The implementation can choose *which subset to test first.* Some subsets may be more likely to cause a failure than others.
- The implementation can choose whether and how to handle *multiple independent failure-inducing inputs*—that is, the case where there are several subsets Δ_i with $test(c'_{✔} \cup \Delta_i) = ✘$. Options include
  - to continue with the first failing subset,
  - to continue with the smallest failing one, or
  - to simplify each individual failing subset.

Our implementation currently goes for the first failing subset only and thus reports only one subset. The reason is economy: it is wiser to fix the first failure before checking for further similar failures.

Undoing changes. Delta Debugging assumes that *failure is monotone:* Once a failure occurs, one cannot make it disappear by adding more “undoing” changes. (Formally, there is no $\bar{\delta}_i$ such that $(\delta_i \circ \bar{\delta}_i)(r) = r$.) As an example, assume a program that processes HTML tags: whenever its input contains only the opening HTML tag, but not the closing one, it fails. In the input `<A></A><B>`, for instance, the HTML tag `<B>` lacks a closing `</B>`.

If we use Delta Debugging to simplify this failure-inducing input, then it may partition the input into `<A>` and `</A><B>`, resulting in the simplified input `<A>`—although in the concrete example, this failure cause was undone by `</A>`; it was `<B>` that had no closing HTML tag. To identify undoing changes, one cannot use *test* alone (this would require testing up to $2^{|c_{✘}|}$ supersets of the minimized test case), so we investigate whether increased precision (Section IV-D) or domain-specific knowledge help in practice.

Program analysis. In the field of general automated debugging, failure-inducing circumstances have almost exclusively been understood as failure-inducing *statements* during a program execution. The most significant method to determine statements relevant for a failure is *program slicing*—either the static form obtained by program analysis [11], [12] or the dynamic form applied to a specific run of the program [13], [14].

The strength of analysis is that several potential failure causes can be eliminated due to lack of data or control dependency. This does not suffice, though, to check whether the remaining potential causes are relevant or not for producing a given failure. Only by experiment (that is, testing) can we prove that some circumstance is relevant—by showing that there is some alteration of the circumstance that makes the failure disappear. When it comes to concrete failures, program analysis and testing are complementary: analysis disproves causality, and testing proves it.

It would be nice to see how far systematic testing and program analysis could work together and whether Delta Debugging could be used to determine failure-inducing statements as well. In our current work, we treat a *program state* (i.e. variables and values) as *internal input* to the remainder of the program and isolate those variables and values which are relevant for producing the failure—just as Delta Debugging isolates external failure-inducing input. The statements in which these variables are set can then be determined as failure-inducing. In both steps, program analysis is most helpful to narrow down the set of potential variables and statements.

Other failure-inducing circumstances. Changing the input of the program is only one means to influence its execution. As stated in Section II-C, a δ_i can stand for any change in the circumstances that influences the execution of the program. Our current work extends Delta Debugging to other failure-inducing circumstances such as executed statements, control predicates or thread schedules.

IX. Conclusion

> *Debugging is still, as it was 30 years ago,*
> *a matter of trial and error.*
>
> — Henry Lieberman, *The Debugging Scandal*

We have shown how the Delta Debugging algorithms simplify and isolate failure-inducing input, based on an automated testing procedure. The method can be (and has been) applied in a number of settings, finding failure-inducing parts in the program invocation (GCC options), in the program input (GCC, fuzz, and Mozilla input), or in the sequence of user interactions (Mozilla user actions).

We recommend that automated test case simplification be an integrated part of automated testing. Each time a test fails, Delta Debugging could be used to simplify and isolate the circumstances of the failure. Given sufficient testing resources and a reasonable choice of changes δ_i that influence the program execution, the algorithms presented in this paper provide simplification and isolation methods that are straight-forward and easy to implement.

In practice, testing and debugging typically come in pairs. However, in previous research on automated debugging, testing has played a very minor role. This is surprising, because re-testing a program under changed circumstances is a common debugging approach—and the only way to prove that the circumstances actually cause the failure. Eventually, we expect that several debugging tasks can in fact be stated as search and minimization problems, based on automated testing—and thus be solved automatically.

Acknowledgments. Mirko Streckenbach provided helpful insights on UNIX internals. Tom Truscott pointed us to the GCC error. Holger Cleve, Jens Krinke and Gregor Snelting provided valuable comments on earlier revisions of this paper. Special thanks go to the ISSTA and TSE reviewers for their constructive comments.

Further information on Delta Debugging is available at

`http://www.st.cs.uni-sb.de/dd/` .

References

[1] “Mozilla web site,” http://www.mozilla.org/.

[2] “Mozilla web site: The Gecko BugAThon,” http://www.mozilla.org/newlayout/bugathon.html.

[3] Andreas Zeller, “Yesterday, my program worked. Today, it does not. Why?,” In Nierstrasz and Lemoine [15], pp. 253–267.

[4] IEEE, New York, *Test Methods for Measuring Conformance to POSIX*, 1991, ANSI/IEEE Standard 1003.3-1991. ISO/IEC Standard 13210-1994.

[5] Marc Vertes, “Xlab—a tool to automate graphical user interfaces,” *Linux Weekly News*, May 1998, Archived as http://lwn.net/980528/a/xlab.html.

[6] B. P. Miller, L. Fredrikson, and B. So, “An empirical study of the reliability of UNIX utilities,” *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[7] Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl, “Fuzz revisited: A re-examination of the reliability of UNIX utilities and services,” Tech. Rep., University of Wisconsin, Computer Science Department, Nov. 1995.

[8] Donald R. Slutz, “Massive stochastic testing of SQL,” in *Proc. of 24rd International Conference on Very Large Data Bases (VLDB’98), New York City, New York, USA*, Ashish Gupta, Oded Shmueli, and Jennifer Widom, Eds. Aug. 1998, pp. 618–622, Morgan Kaufmann.

[9] David B. Whalley, “Automatic isolation of compiler errors,” *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1648–1659, 1994.

[10] Andreas Zeller and Gregor Snelting, “Unified versioning through feature logic,” *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 4, pp. 398–441, Oct. 1997.

[11] M. Weiser, “Programmers use slices when debugging,” *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982.

[12] Frank Tip, “A survey of program slicing techniques,” *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, Sept. 1995.

[13] Hiralal Agrawal and Joseph R. Horgan, “Dynamic program slicing,” in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI)*, White Plains, New York, June 1990, vol. 25(6) of *ACM SIGPLAN Notices*, pp. 246–256.

[14] Tibor Gyimóthy, Árpád Beszédes, and István Forgács, “An efficient relevant slicing method for debugging,” In Nierstrasz and Lemoine [15], pp. 303–321.

[15] Oscar Nierstrasz and Michel Lemoine, Eds., *Proc. ESEC/FSE’99 – 7th European Software Engineering Conference / 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 1687 of *Lecture Notes in Computer Science*, Toulouse, France, Sept. 1999. Springer-Verlag.

Andreas Zeller received his PhD in computer science from Technische Universität Braunschweig, Germany in 1997. He is currently a professor of computer science at Universität des Saarlandes, Saarbrücken, Germany. His general research interest is in program comprehension, especially program analysis, automated debugging, and software visualization – and how to make these techniques applicable to real programs with real problems. He is a member of the IEEE computer society, the ACM, and the GI.

Ralf Hildebrandt received his computer science diploma from Technische Universität Braunschweig in 2000, after conducting the actual diploma work at Universität Passau. His diploma thesis on the minimization of failure-inducing input was awarded the Ernst Denert Award for the best German diploma thesis in software engineering. He is currently a senior software engineer at DeTeLine, Berlin, Germany. He is a member of the GI.