CIS 547
Software Analysis

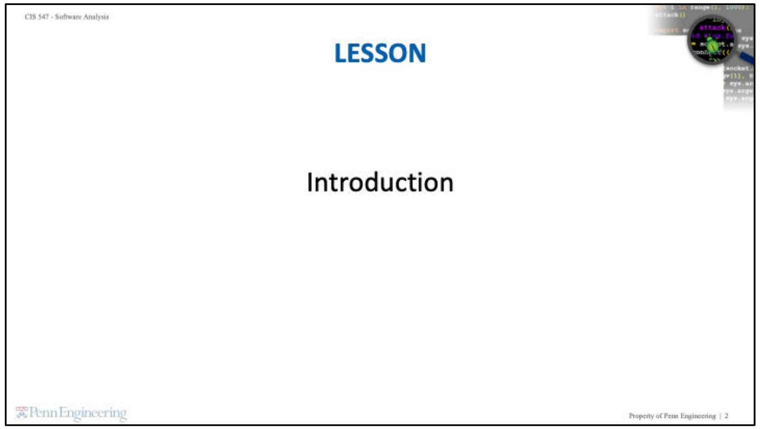
Delta Debugging

Mayur Naik

Penn Engineering

Often when debugging, we find ourselves with the problem of having an input that crashes a program but not knowing what aspect of the input is causing the program's failure. For example, a webpage with hundreds of lines of HTML crashes a browser, or a random sequence of keystrokes crashes a smartphone app. Isolating the cause of the failure would be enormously helpful in finding what change needs to be made to the program's code.

One automated technique for paring down large failing inputs is delta debugging. Delta debugging is based on the scientific method: hypothesize, experiment, and refine. By selectively and systematically removing portions of the input, delta debugging automatically removes irrelevant information from a failing test case in order to attain a "minimal" bug-inducing input.



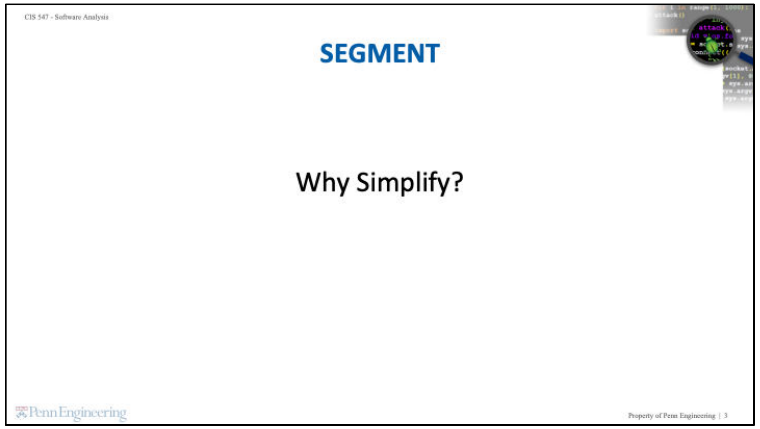
CIS 547 - Software Analysis

LESSON

Introduction

Penn Engineering

Property of Penn Engineering | 2

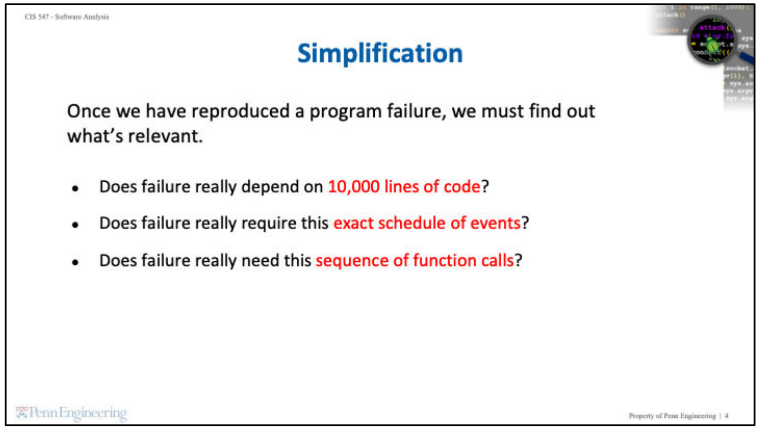


CIS 547 - Software Analysis

SEGMENT

Why Simplify?

Property of Penn Engineering | 3



CIS 547 - Software Analysis

Simplification

Once we have reproduced a program failure, we must find out what's relevant.

- Does failure really depend on 10,000 lines of code?
- Does failure really require this exact schedule of events?
- Does failure really need this sequence of function calls?

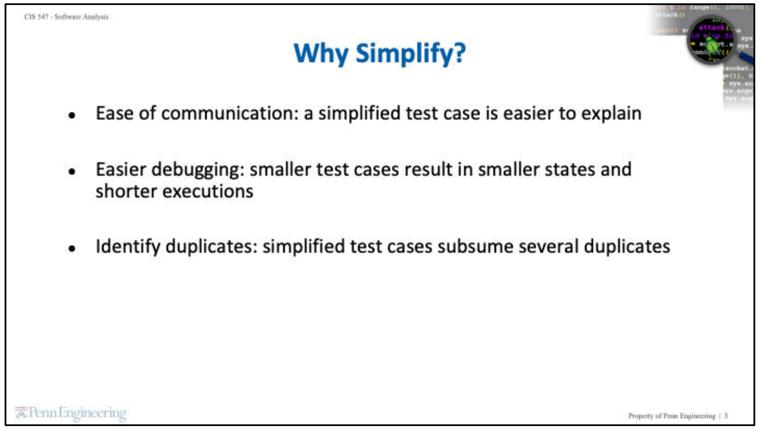
Property of Penn Engineering | 4

A typical bug report contains a lot of information that the developer can use to reproduce the program failure. Once we have reproduced a program failure, we must find out what information is relevant. For instance,

Does the failure really depend on 10,000 lines of code?

Does the failure really require this exact schedule of events?

Does the failure really need this sequence of function calls?



Why Simplify?

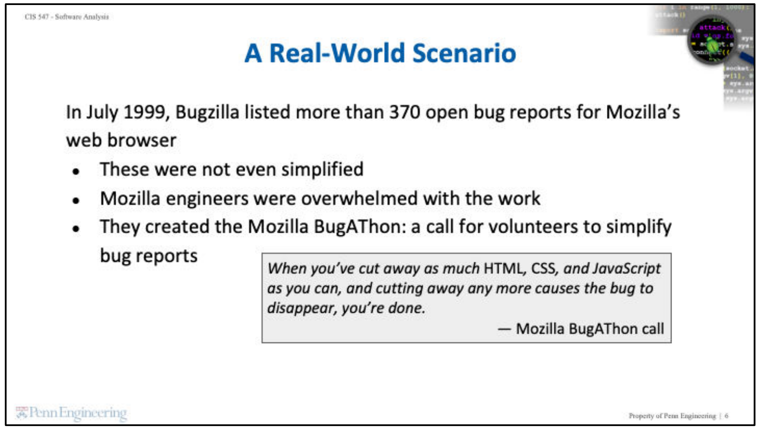
- Ease of communication: a simplified test case is easier to explain
- Easier debugging: smaller test cases result in smaller states and shorter executions
- Identify duplicates: simplified test cases subsume several duplicates

Simplifying the information in a bug report down to only what is relevant is important for several reasons.

First is ease of communication: a simplified test case is easier to communicate to members of the development and testing team.

Second is that simpler test cases lead to easier debugging: a smaller test case results in smaller states and shorter executions.

Third is that it allows us to identify and collapse duplicate issues: a simplified test case can subsume test cases in several bug reports.



A Real-World Scenario

In July 1999, Bugzilla listed more than 370 open bug reports for Mozilla's web browser

- These were not even simplified
- Mozilla engineers were overwhelmed with the work
- They created the Mozilla BugAThon: a call for volunteers to simplify bug reports

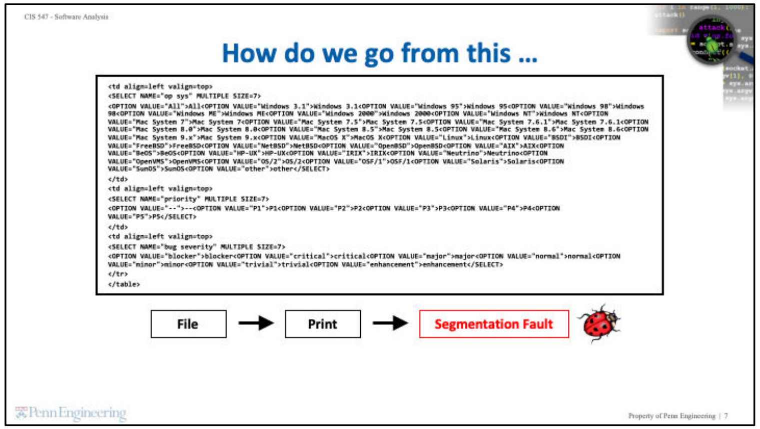
> *When you've cut away as much* HTML, CSS, *and* JavaScript *as you can, and cutting away any more causes the bug to disappear, you're done.*
>
> — Mozilla BugAThon call

Let's look at a real-world scenario which should help motivate the necessity for bug minimization.

In July of 1999, Bugzilla, the Mozilla bug database, had over 370 unresolved bug reports for Mozilla's web browser. These reports weren't even simplified, and the bug queue was growing by the day. Mozilla's engineers became overwhelmed with the workload.

Under this pressure, the project manager sent out a call for volunteers for the Mozilla BugAThon: volunteers to help process the bug reports. Their goal was to turn each bug report into a minimal test case, in which each part of the input is significant in reproducing the failure. The volunteers were even rewarded with perks for their work: volunteers who simplified 5 reports would be invited to the launch party, and those who simplified 20 reports would receive a T-shirt signed by the engineering team.

Clearly, Mozilla would have benefitted from an automated bug minimization process here!



CIS 547 - Software Analysis

How do we go from this ...

```
<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>
```

PennEngineering

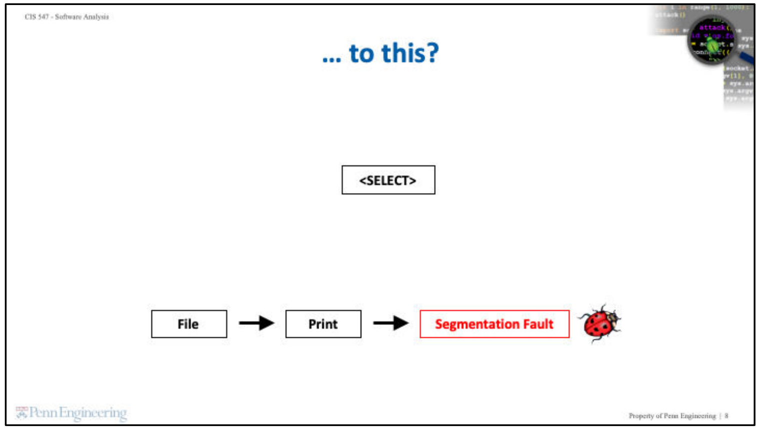
Property of Penn Engineering | 7

Let’s look at a concrete bug report in the Mozilla bug database.

Consider this HTML page. Loading this page using a certain version of Mozilla’s web browser and printing it causes a segmentation fault. Somewhere in this HTML input is something that makes the browser fail. But how do we find it?

If we were the developers of the Mozilla web browser that crashes on this input, we would want the simplest HTML input that still causes the crash.

So how do we go from this large input to ...



CIS 547 - Software Analysis

... to this?

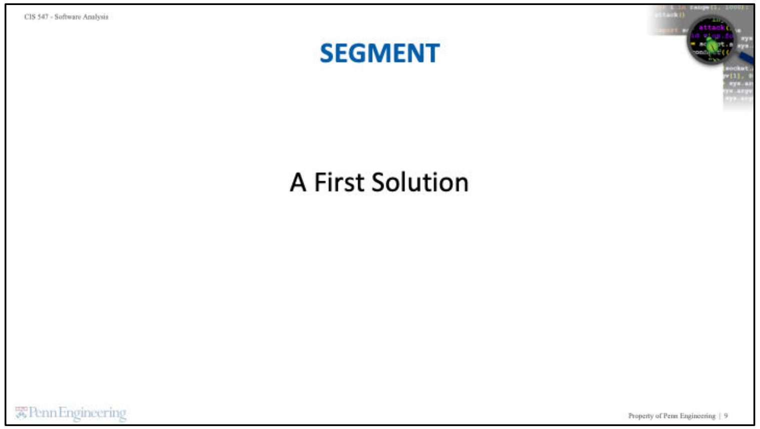
```
<SELECT>
```

File
Print
Segmentation Fault

PennEngineering

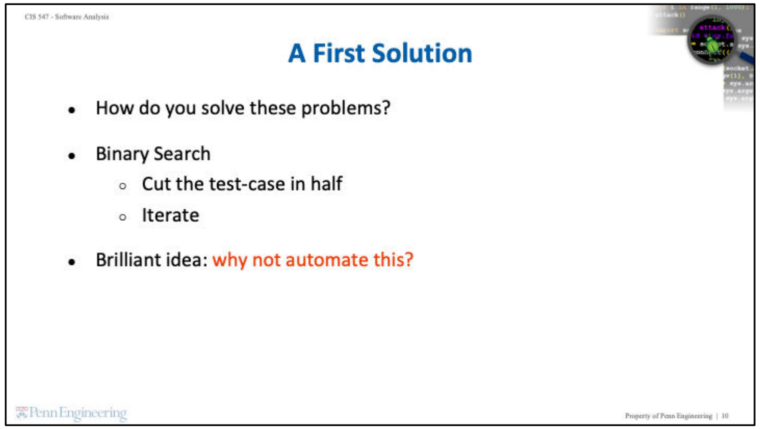
Property of Penn Engineering | 8

this simple input -- a mere select tag -- that still causes the crash?



SEGMENT

A First Solution

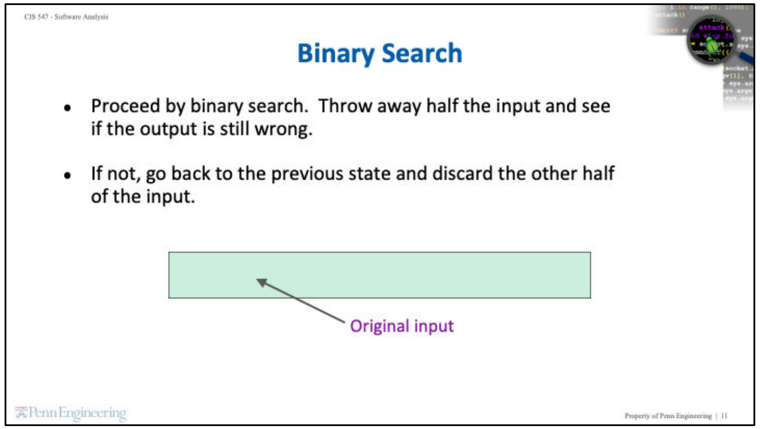


A First Solution

- How do you solve these problems?
- Binary Search
  - Cut the test-case in half
  - Iterate
- Brilliant idea: why not automate this?

What do we as humans do in order to minimize test cases?

One possibility is that we might use a binary search, cutting the test case in two and testing each half of the input separately. We could even iterate this procedure to shrink the input as much as possible. Even better, binary search is a process that can be easily automated for large test cases!

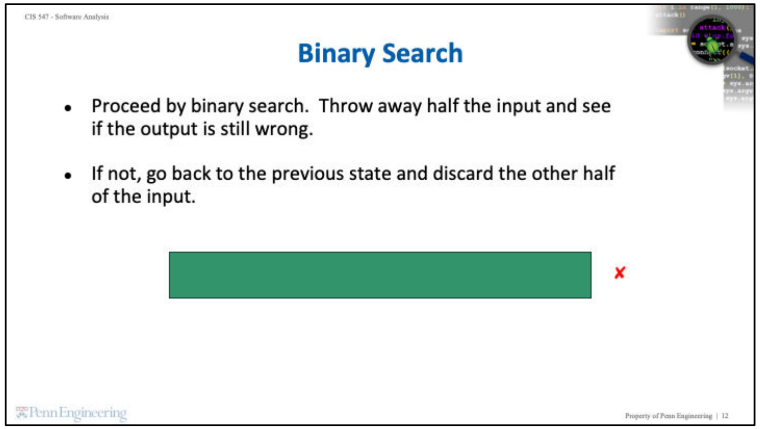


Binary Search

- 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.

Original input

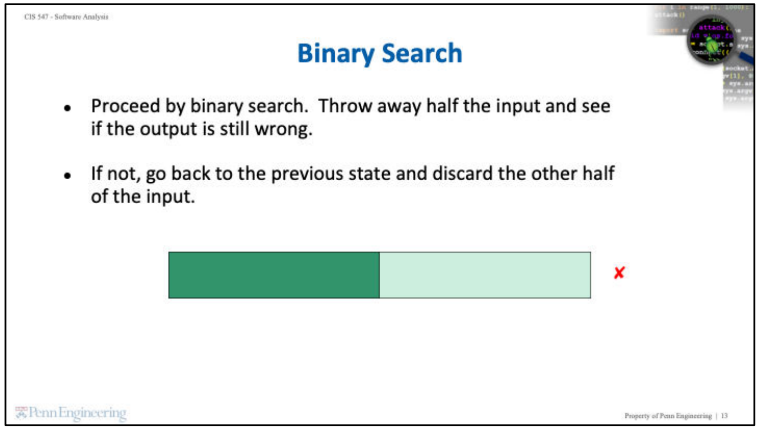
Let's see how this application of binary search might work.



Binary Search

- 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.

This bar here represents the original failure-inducing input to a program.

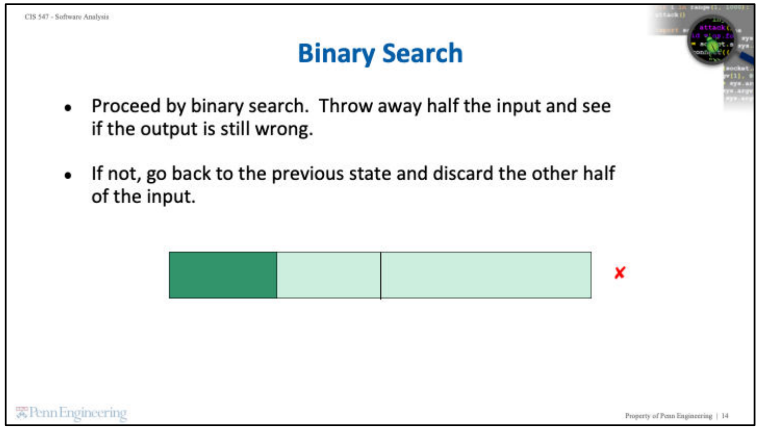


Binary Search

- 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.

✘

If one half of the input causes the program to fail, then we can eliminate the second half of the input and attain a smaller, failure-inducing input.

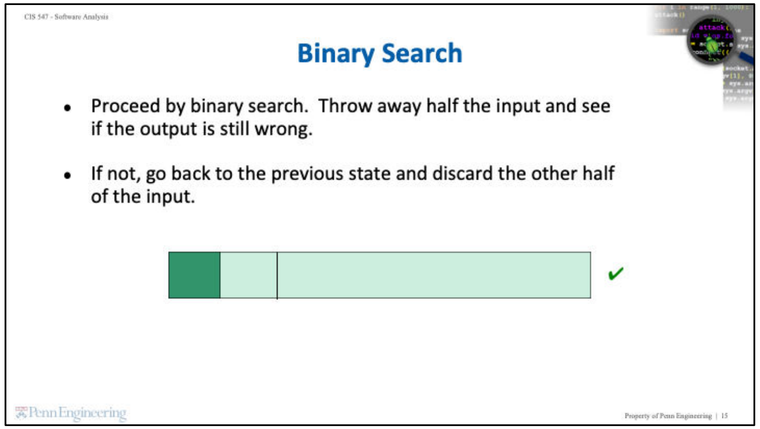


Binary Search

- 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.

✘

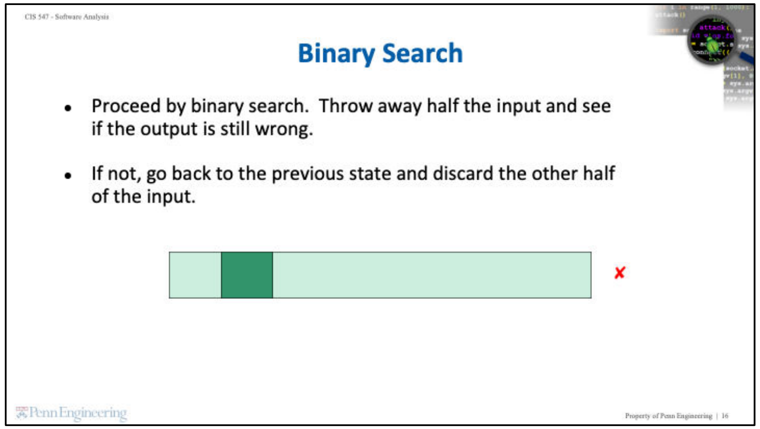
We can repeat the procedure on the new, smaller failing input; in this case, the first half of the halved input causes the program to fail, so we can throw away the second half. We are left with an input that induces a failure but which is a quarter of the size of the original input.



Binary Search

- 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.

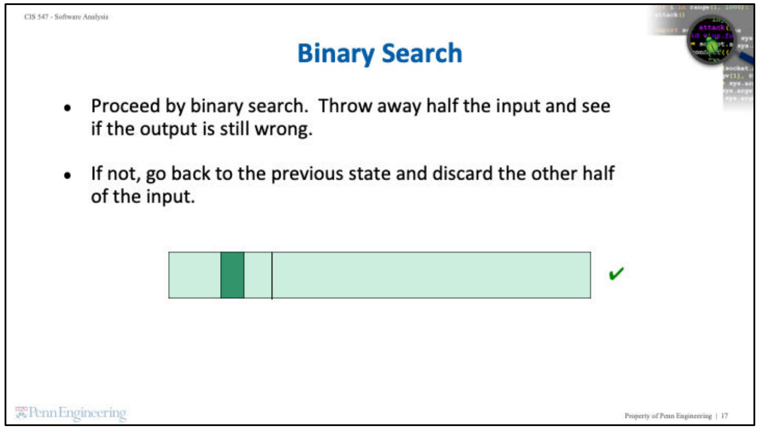
Repeating the procedure, we might find that the first half of the new input does not crash the program ...



Binary Search

- 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.

... but that the second half does cause the program to fail. In this case, we’d remove the first half and keep the second half to obtain yet a smaller failure-inducing input.



CIS 547 - Software Analysis

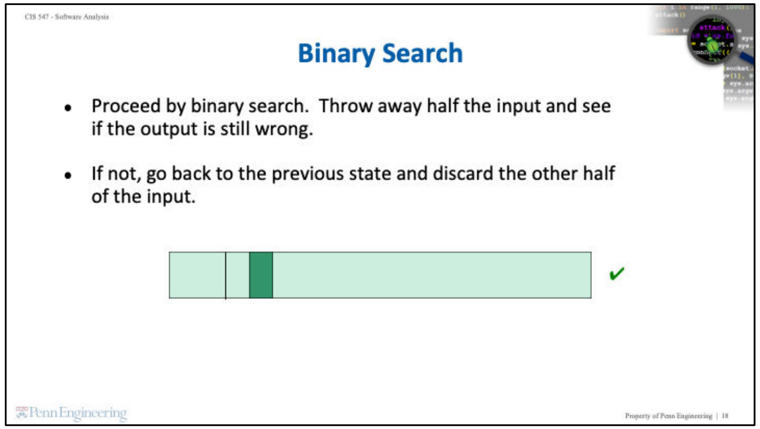
Binary Search

- 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.

PennEngineering

Property of Penn Engineering | 17

Iterating again, we might find that the first half of the new input does not crash the program ...



CIS 547 - Software Analysis

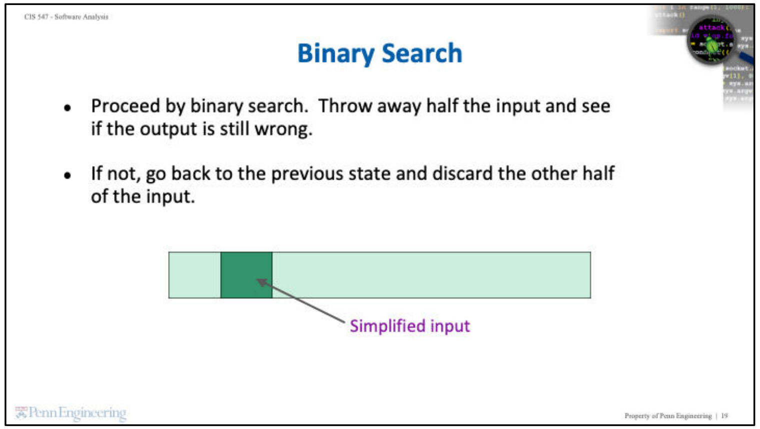
Binary Search

- 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.

PennEngineering

Property of Penn Engineering | 18

... and so does the second half.



CIS 547 - Software Analysis

Binary Search

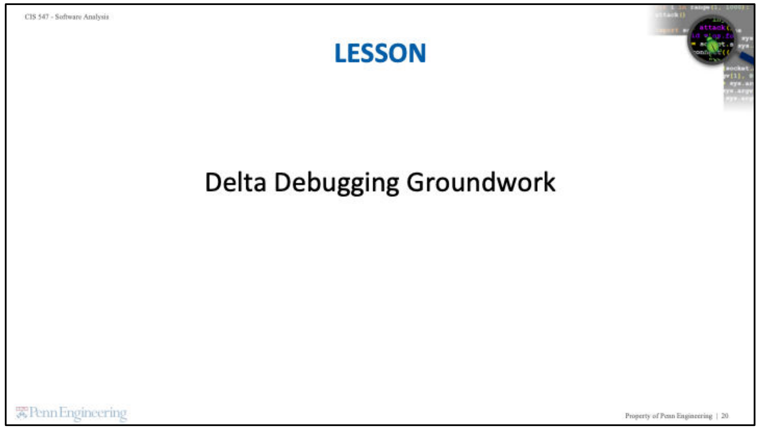
- Proceed by binary search. Throw away half the input and see if the output is still wrong.
- If not, go back to the previous state and discard the other half of the input.

Simplified input

Penn Engineering

Property of Penn Engineering | 19

In this case, binary search can proceed no further. The simplified input is this dark green portion of the bar, one-eighth the size of the original input, which is good progress! While this is a good first solution, however, we shall see next that a naïve binary search is not always adequate to sufficiently minimize the input.



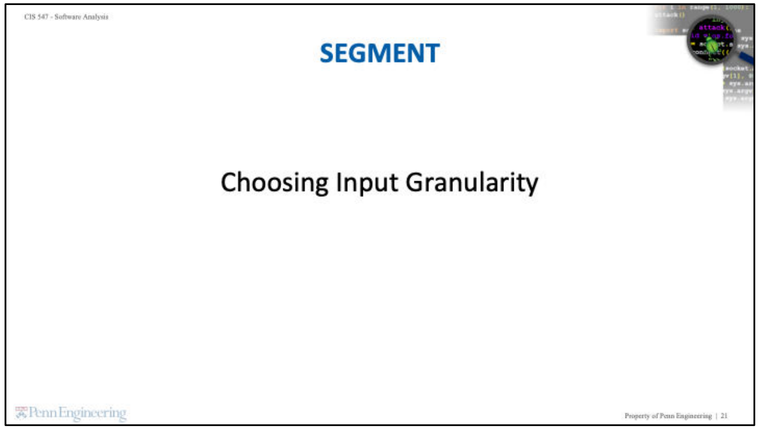
CIS 547 - Software Analysis

LESSON

Delta Debugging Groundwork

Penn Engineering

Property of Penn Engineering | 20



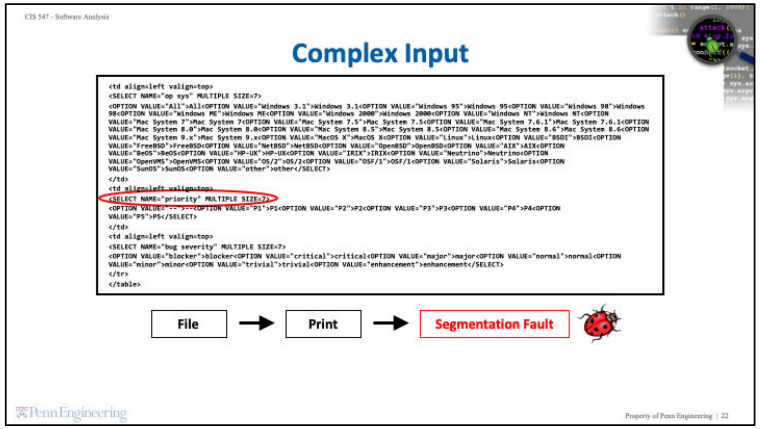
CIS 547 - Software Analysis

SEGMENT

Choosing Input Granularity

Penn Engineering

Property of Penn Engineering | 21



CIS 547 - Software Analysis

Complex Input

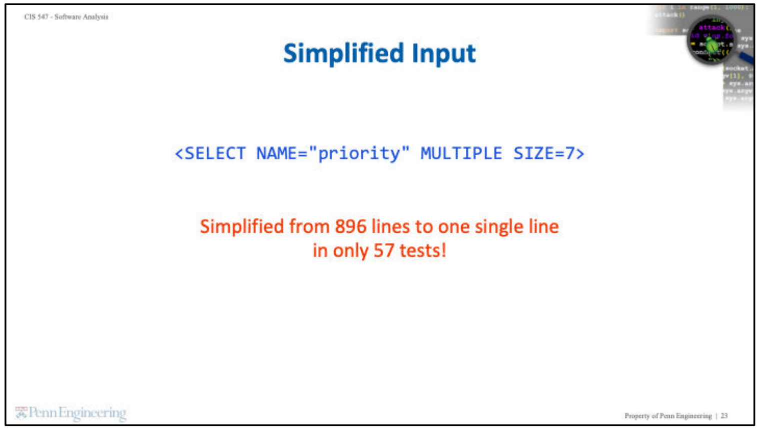
```
<td align=left valign=top>
<SELECT NAME="op sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows 3.1<OPTION VALUE="Windows 95">Windows 95<OPTION VALUE="Windows 98">Windows
98<OPTION VALUE="Windows ME">Windows ME<OPTION VALUE="Windows 2000">Windows 2000<OPTION VALUE="Windows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

Penn Engineering

Property of Penn Engineering | 22

We can apply the binary search algorithm to minimize the number of lines in the HTML input we saw earlier: the one that crashed Mozilla's web browser. We'll assume line granularity of the input for this purpose; that is, we only partition the input at line breaks.

The algorithm outputs the following line …

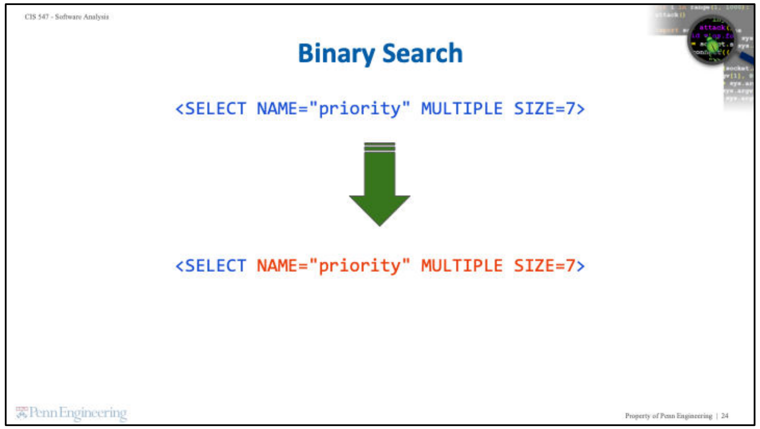


Simplified Input

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

Simplified from 896 lines to one single line
in only 57 tests!

simplifying from 896 lines in the original input to this single line, in only 57 tests.

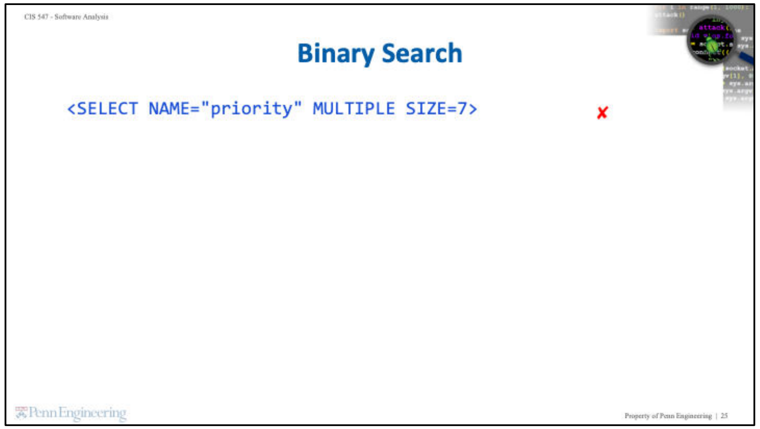


Binary Search

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

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

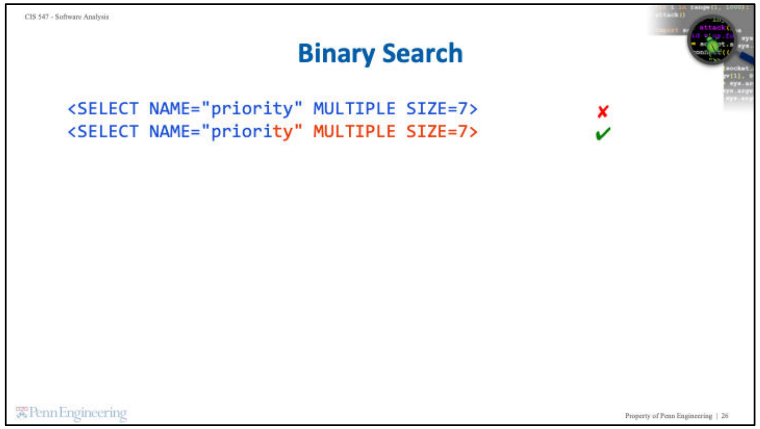
Suppose that we wish to further simplify this input using character-level granularity to obtain the desired output comprising only the <SELECT> tag. Let’s see how the binary search algorithm works this time.



Binary Search

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

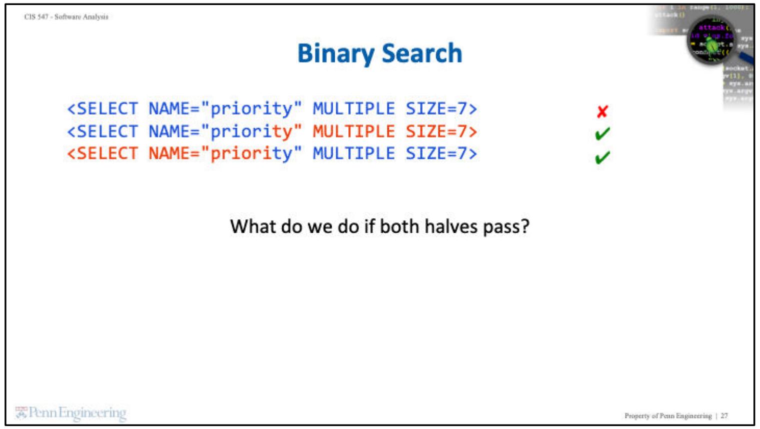
The initial input consisting of the entire line causes the browser to crash.



Binary Search

```
<SELECT NAME="priority" MULTIPLE SIZE=7>          ✘
<SELECT NAME="priority" MULTIPLE SIZE=7>          ✓
```

The first half of the line doesn’t cause the browser to crash.

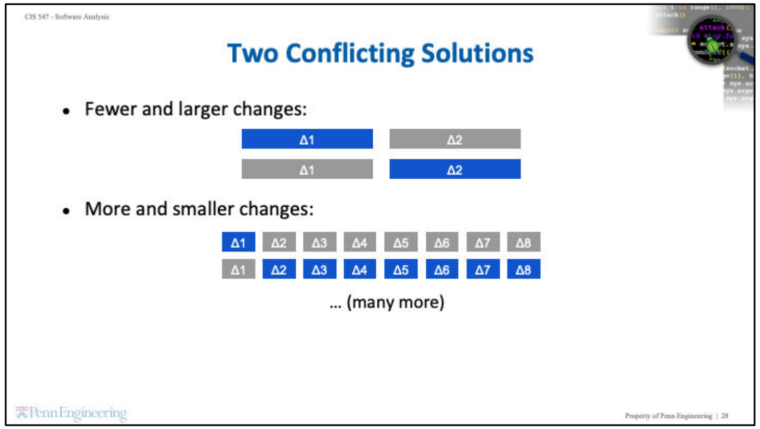


Binary Search

```
<SELECT NAME="priority" MULTIPLE SIZE=7>   ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>   ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>   ✓
```

What do we do if both halves pass?

And the second half of the line also doesn't cause the browser to crash. At this point, binary search says we're stuck, since neither half of the input induces a failure on its own. Is there some other way we can minimize the input?



Two Conflicting Solutions

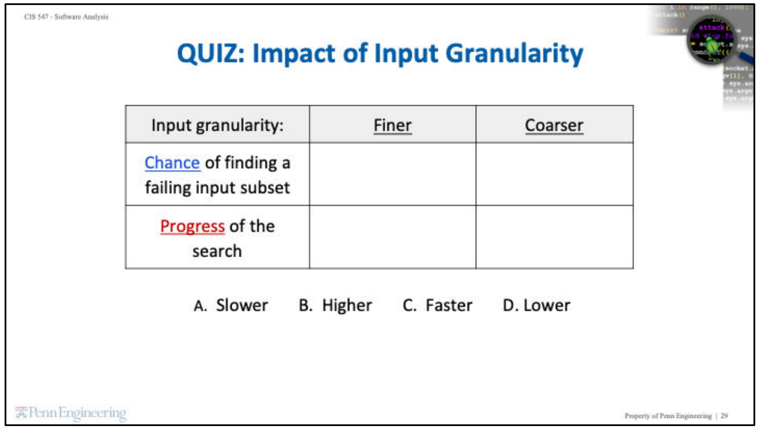
- Fewer and larger changes:

- More and smaller changes:

… (many more)

Let's generalize the binary search procedure a bit. Instead of requiring ourselves to divide inputs strictly in half at each iteration, we could allow ourselves more granularity in dividing our input.

Perhaps we could divide up our input into many (possibly disconnected) subsets at an iteration and only keep those which are required to cause a failure. In particular, we can break up the input into blocks of any size, called "changes" from the original input. (The traditional use of the Greek letter "delta" for change is the origin of the name "delta debugging.") We can then use subsets formed from these blocks. Perhaps we just use a single block, perhaps we use several blocks concatenated together, or perhaps we use noncontiguous blocks (for example, block delta-1 and block delta-8) to form subsets for testing.



CIS 547 - Software Analysis

QUIZ: Impact of Input Granularity

Input granularity:	Finer	Coarser
Chance of finding a failing input subset		
Progress of the search		

A. Slower B. Higher C. Faster D. Lower

Penn Engineering

Property of Penn Engineering | 29

{QUIZ SLIDE}

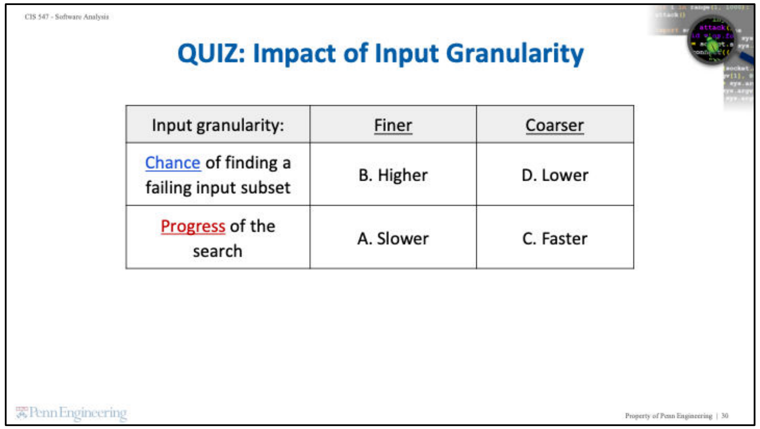
This gives us two opposing strategies with their own strengths and weaknesses.

Take a moment to consider what might happen if we allow the granularity of the input changes we use to be finer or coarser.

Finer granularity means our input is divided into more, smaller blocks; coarser granularity means our input is divided into fewer, larger blocks.

What would happen to the chance of finding a failing subset of the input? And how much progress would we make if we found a failing subset of the input?

Fill in each box with the appropriate letter: A for slower progress, B for higher chance of finding a failing subset, C for faster progress, and D for lower chance of finding a failing subset.



CIS 547 - Software Analysis

QUIZ: Impact of Input Granularity

Input granularity:	Finer	Coarser
Chance of finding a failing input subset	B. Higher	D. Lower
Progress of the search	A. Slower	C. Faster

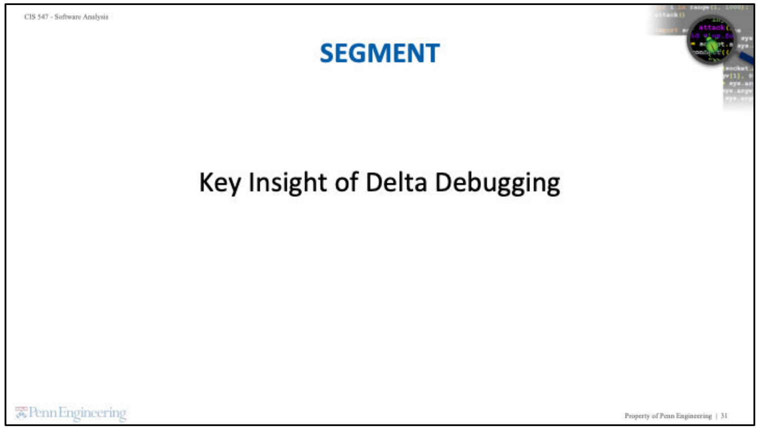
Penn Engineering

Property of Penn Engineering | 30

{SOLUTION SLIDE}

By testing subsets made up of larger blocks (coarser granularity), we lower our chance of finding some subset of the blocks that fails a test, but we have fewer subsets that we need to test. Additionally, upon finding a subset of blocks that fails, we can eliminate a large portion of the input string. This means our progress toward a minimal test case is faster.

On the other hand, by testing subsets made up of smaller changes (finer granularity), we have more subsets that we have to test; and upon finding a subset of changes which causes failure, we typically can only remove small portions of the input string. These both slow our progress towards finding a minimal failing test case. However, the tradeoff is that by testing more subsets, we increase our chance of finding a smaller subset that actually does cause a failure. Indeed, we could go so far as to making the granularity of the input changes one character in size, which would guarantee we find the minimum failing test case, but this strategy in the worst case would take exponential time in the length of the input.



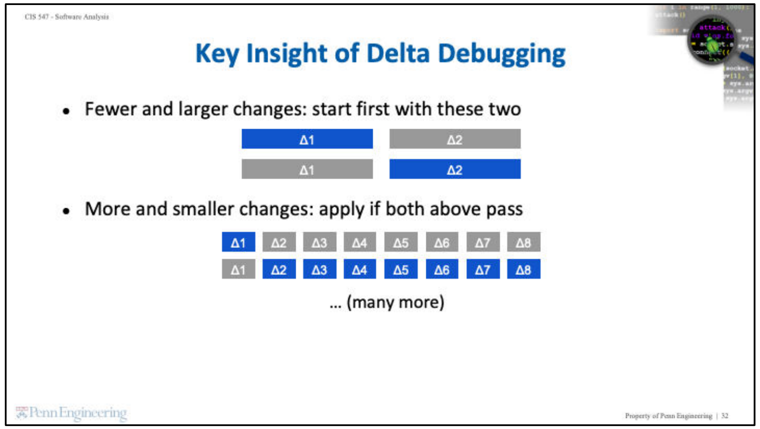
CIS 547 - Software Analysis

SEGMENT

Key Insight of Delta Debugging

Penn Engineering

Property of Penn Engineering | 31



CIS 547 - Software Analysis

Key Insight of Delta Debugging

- Fewer and larger changes: start first with these two

Δ1 Δ2

Δ1 Δ2

- More and smaller changes: apply if both above pass

Δ1 Δ2 Δ3 Δ4 Δ5 Δ6 Δ7 Δ8

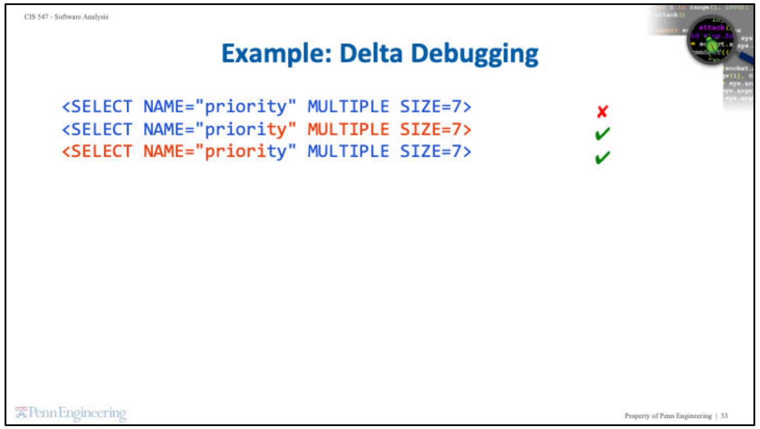
Δ1 Δ2 Δ3 Δ4 Δ5 Δ6 Δ7 Δ8

… (many more)

Penn Engineering

Property of Penn Engineering | 32

The key insight of delta debugging is that it combines the best of both approaches. At first, it divides the input into larger blocks, and it tests all subsets of these changes for failure. As the algorithm becomes unable to find subsets that fail the test, delta debugging increases the granularity of its changes, testing subsets formed from smaller blocks for failure.



CIS 547 - Software Analysis

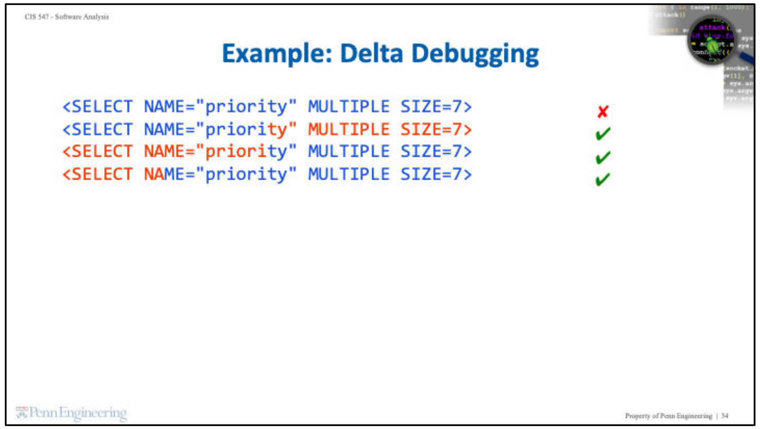
Example: Delta Debugging

```
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✘
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
```

Penn Engineering

Property of Penn Engineering | 33

Let’s see how delta debugging would proceed with our example from earlier. Recall that neither half of the original input string caused a crash in Mozilla’s browser.



CIS 547 - Software Analysis

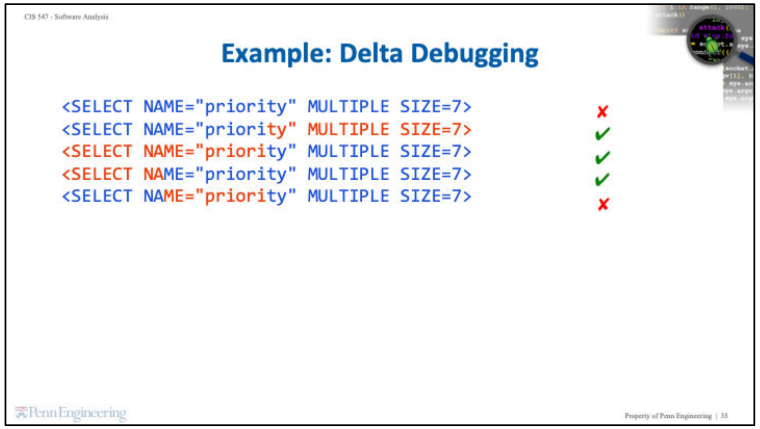
Example: Delta Debugging

```
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✘
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
```

Penn Engineering

Property of Penn Engineering | 34

Using delta debugging, we increase the granularity of the blocks we will use to create subsets, by decreasing their size by a factor of two. We first test the subset formed from the second, third, and fourth blocks: this subset doesn’t cause a crash since it does not include the <SELECT> tag.



CIS 547 - Software Analysis

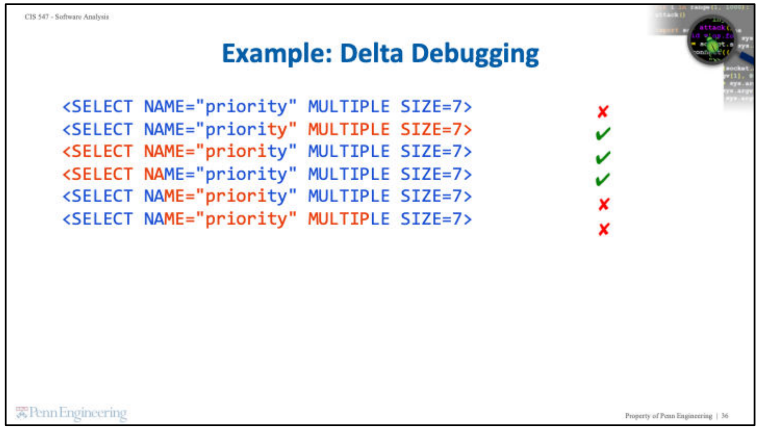
Example: Delta Debugging

```
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✗
```

Penn Engineering

Property of Penn Engineering | 35

Next we test the subset formed from the first, third, and fourth blocks. This does cause a crash, since it includes the <SELECT> tag, so we can eliminate the second block from consideration altogether.



CIS 547 - Software Analysis

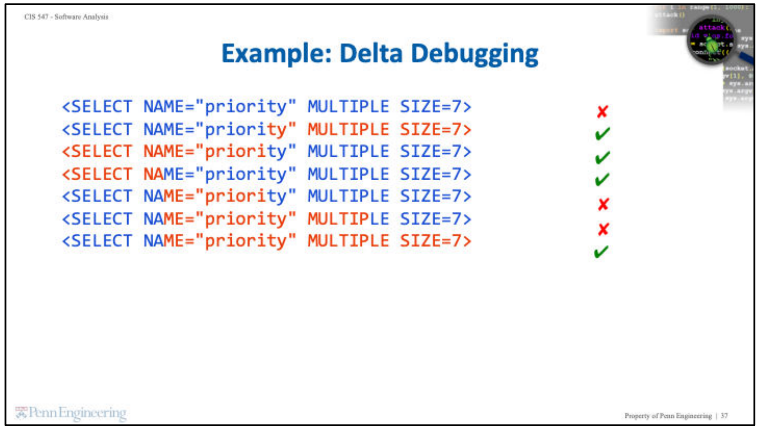
Example: Delta Debugging

```
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✗
```

Penn Engineering

Property of Penn Engineering | 36

Next, let’s see what happens if we keep only the first and fourth blocks. Again, this causes a crash, since it includes the <SELECT> tag, so we can eliminate the third block from consideration.



CIS 547 - Software Analysis

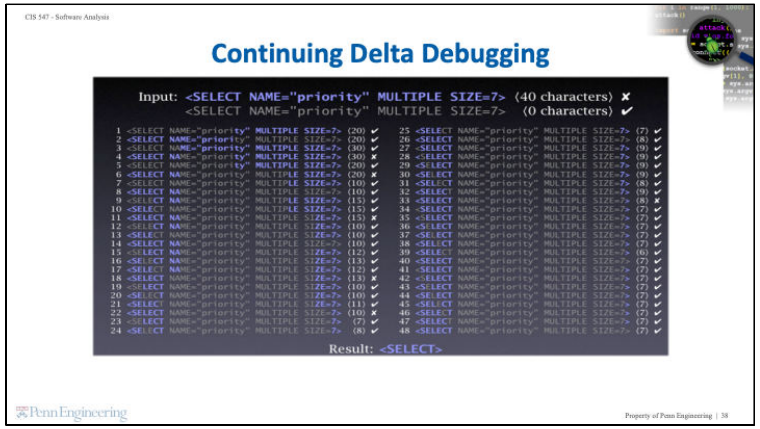
Example: Delta Debugging

```
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✘
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✔
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✔
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✔
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✘
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✘
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✔
```

Penn Engineering

Property of Penn Engineering | 37

Removing the fourth block causes the input to pass the test, as the closing bracket of the <SELECT> tag is missing in this input. So we would end up keeping the first and fourth blocks and increasing the granularity before continuing to test subsets ...



CIS 547 - Software Analysis

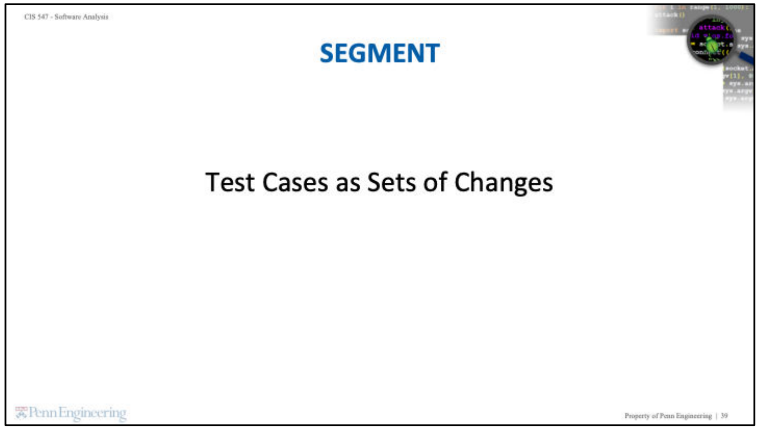
Continuing Delta Debugging

```
Input: <SELECT NAME="priority" MULTIPLE SIZE=7> (40 characters) ✘
       <SELECT NAME="priority" MULTIPLE SIZE=7>  (0 characters) ✔

 1 <SELECT NAME="priority" MULTIPLE SIZE=7> (20) ✔   25 <SELECT NAME="priority" MULTIPLE SIZE=7> (7) ✔
 2 <SELECT NAME="priority" MULTIPLE SIZE=7> (20) ✔   26 <SELECT NAME="priority" MULTIPLE SIZE=7> (8) ✔
 3 <SELECT NAME="priority" MULTIPLE SIZE=7> (30) ✔   27 <SELECT NAME="priority" MULTIPLE SIZE=7> (9) ✔
 4 <SELECT NAME="priority" MULTIPLE SIZE=7> (30) ✘   28 <SELECT NAME="priority" MULTIPLE SIZE=7> (9) ✔
 5 <SELECT NAME="priority" MULTIPLE SIZE=7> (20) ✔   29 <SELECT NAME="priority" MULTIPLE SIZE=7> (9) ✔
 6 <SELECT NAME="priority" MULTIPLE SIZE=7> (20) ✘   30 <SELECT NAME="priority" MULTIPLE SIZE=7> (9) ✔
 7 <SELECT NAME="priority" MULTIPLE SIZE=7> (10) ✔   31 <SELECT NAME="priority" MULTIPLE SIZE=7> (8) ✔
 8 <SELECT NAME="priority" MULTIPLE SIZE=7> (10) ✔   32 <SELECT NAME="priority" MULTIPLE SIZE=7> (9) ✔
 9 <SELECT NAME="priority" MULTIPLE SIZE=7> (15) ✔   33 <SELECT NAME="priority" MULTIPLE SIZE=7> (8) ✘
10 <SELECT NAME="priority" MULTIPLE SIZE=7> (15) ✔   34 <SELECT NAME="priority" MULTIPLE SIZE=7> (7) ✔
11 <SELECT NAME="priority" MULTIPLE SIZE=7> (15) ✘   35 <SELECT NAME="priority" MULTIPLE SIZE=7> (7) ✔
12 <SELECT NAME="priority" MULTIPLE SIZE=7> (10) ✔   36 <SELECT NAME="priority" MULTIPLE SIZE=7> (7) ✔
13 <SELECT NAME="priority" MULTIPLE SIZE=7> (10) ✔   37 <SELECT NAME="priority" MULTIPLE SIZE=7> (7) ✔
14 <SELECT NAME="priority" MULTIPLE SIZE=7> (10) ✔   38 <SELECT NAME="priority" MULTIPLE SIZE=7> (7) ✔
15 <SELECT NAME="priority" MULTIPLE SIZE=7> (12) ✔   39 <SELECT NAME="priority" MULTIPLE SIZE=7> (6) ✔
16 <SELECT NAME="priority" MULTIPLE SIZE=7> (13) ✔   40 <SELECT NAME="priority" MULTIPLE SIZE=7> (7) ✔
17 <SELECT NAME="priority" MULTIPLE SIZE=7> (12) ✔   41 <SELECT NAME="priority" MULTIPLE SIZE=7> (7) ✔
18 <SELECT NAME="priority" MULTIPLE SIZE=7> (13) ✘   42 <SELECT NAME="priority" MULTIPLE SIZE=7> (7) ✔
19 <SELECT NAME="priority" MULTIPLE SIZE=7> (10) ✔   43 <SELECT NAME="priority" MULTIPLE SIZE=7> (7) ✔
20 <SELECT NAME="priority" MULTIPLE SIZE=7> (10) ✔   44 <SELECT NAME="priority" MULTIPLE SIZE=7> (7) ✔
21 <SELECT NAME="priority" MULTIPLE SIZE=7> (11) ✔   45 <SELECT NAME="priority" MULTIPLE SIZE=7> (7) ✔
22 <SELECT NAME="priority" MULTIPLE SIZE=7> (10) ✘   46 <SELECT NAME="priority" MULTIPLE SIZE=7> (7) ✔
23 <SELECT NAME="priority" MULTIPLE SIZE=7>  (7) ✔   47 <SELECT NAME="priority" MULTIPLE SIZE=7> (7) ✔
24 <SELECT NAME="priority" MULTIPLE SIZE=7>  (8) ✔   48 <SELECT NAME="priority" MULTIPLE SIZE=7> (7) ✔

Result: <SELECT>
```

Penn Engineering

Property of Penn Engineering | 38

... until we eventually end up with the minimal failing input, comprising only the <SELECT> tag, after 48 iterations.



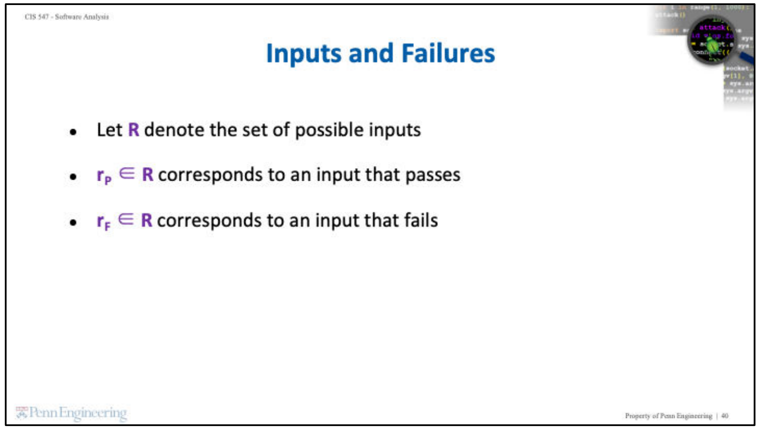
CIS 547 - Software Analysis

SEGMENT

Test Cases as Sets of Changes

Penn Engineering

Property of Penn Engineering | 39



CIS 547 - Software Analysis

Inputs and Failures

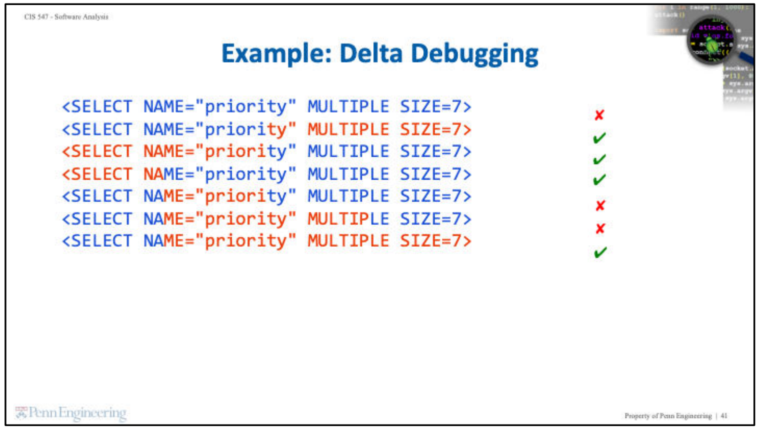
- Let **R** denote the set of possible inputs
- $\mathbf{r_P} \in \mathbf{R}$ corresponds to an input that passes
- $\mathbf{r_F} \in \mathbf{R}$ corresponds to an input that fails

Penn Engineering

Property of Penn Engineering | 40

Now that we've seen an example of how delta debugging would work in practice, let's formally define the algorithm so that we can analyze its properties and prove that it will work as expected.

Let R be the set of all possible inputs that we wish the delta debugging algorithm to consider. We'll use r_P to denote an element of R on which the program passes, and r_F to denote an element of R on which the program fails.



CIS 547 - Software Analysis

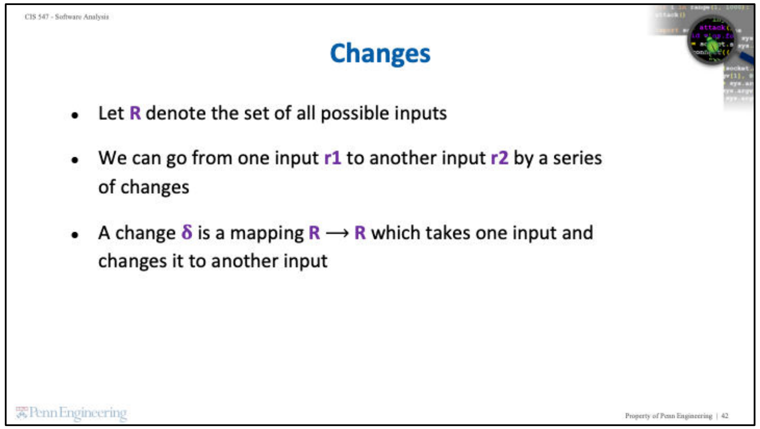
Example: Delta Debugging

```
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✓
```

Penn Engineering

An example of r_p is any of the following passing inputs, such as [hand-circle any input with a tick mark].

An example of r_f is any of the following failing inputs, such as [hand-circle any input with a cross mark].



CIS 547 - Software Analysis

Changes

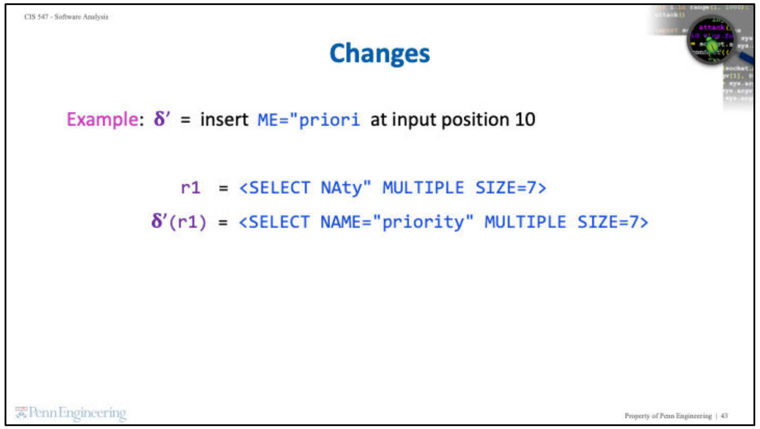
- Let **R** denote the set of all possible inputs
- We can go from one input **r1** to another input **r2** by a series of changes
- A change **δ** is a mapping **R** ⟶ **R** which takes one input and changes it to another input

Penn Engineering

The key building block in the delta debugging algorithm is the concept of a change, which is how one input is transformed into another.

Formally, a change is a mapping from the set of all test inputs to itself. In other words, it's a function that takes a test input r_1 and returns another test input r_2.

[In the Mozilla example from earlier, applying δ means to expand a trivial (empty) HTML input to the full failure-inducing HTML page.]



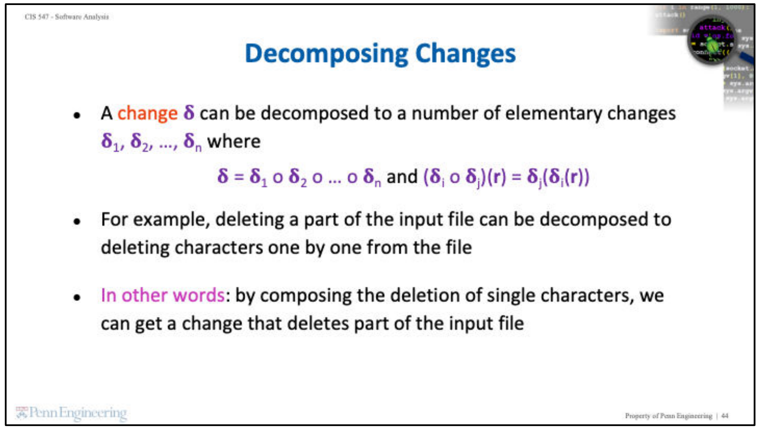
Changes

Example: δ' = insert `ME="priori` at input position 10

```
    r1  = <SELECT NAty" MULTIPLE SIZE=7>
δ'(r1)  = <SELECT NAME="priority" MULTIPLE SIZE=7>
```

As an example, the operation of inserting the string ME=”priori between the 10th and 11th character positions of the input would be an example of a change function relevant to the Mozilla example from before.

Other examples of change functions are the operation of concatenating a semicolon at the end of a string, removing the first character of a nonempty string, and reversing the order of a string. Even the function that simply returns its input string is a change: it serves as the identity change function.



Decomposing Changes

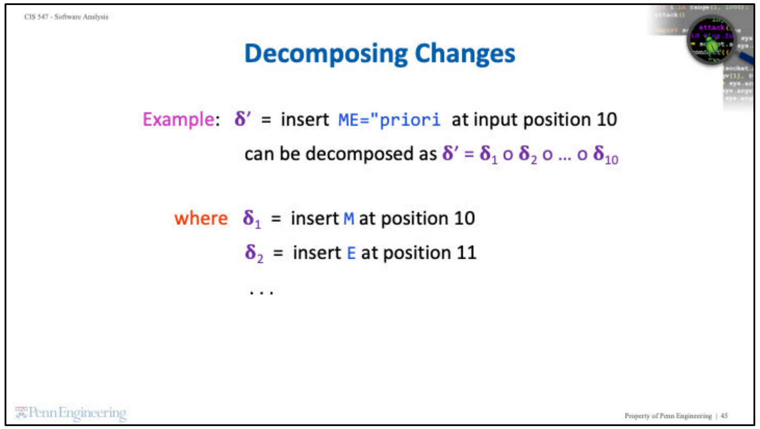
- A change δ can be decomposed to a number of elementary changes $\delta_1, \delta_2, \ldots, \delta_n$ where

$$\delta = \delta_1 \circ \delta_2 \circ \ldots \circ \delta_n \text{ and } (\delta_i \circ \delta_j)(r) = \delta_j(\delta_i(r))$$

- For example, deleting a part of the input file can be decomposed to deleting characters one by one from the file
- In other words: by composing the deletion of single characters, we can get a change that deletes part of the input file

We next introduce the concept of decomposing a change function into a number of elementary change functions such that applying each elementary change in order to an input r has the same effect as applying the original change to that input r all at once.

For example, suppose deleting a line from an input file results in a failure. We can decompose this deletion to a sequence of individual character deletions.



Decomposing Changes

Example: δ' = insert `ME="priori` at input position 10

can be decomposed as $\delta' = \delta_1 \circ \delta_2 \circ \ldots \circ \delta_{10}$

where δ_1 = insert `M` at position 10

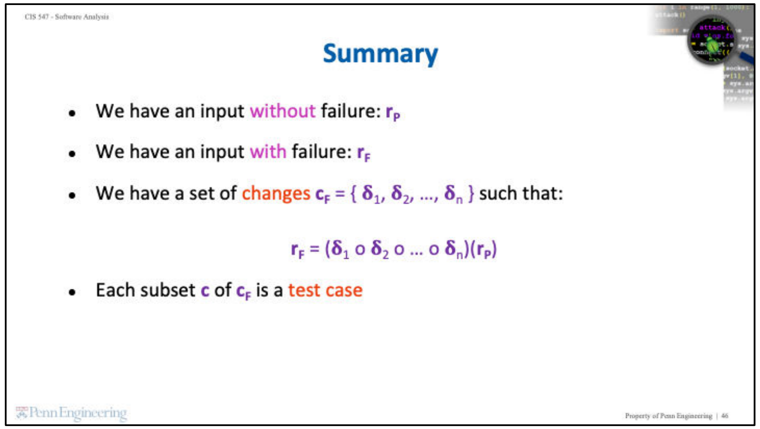
δ_2 = insert `E` at position 11

. . .

Looking again at our Mozilla example from before, we can decompose the change denoted by delta', which represents inserting this string at input position 10, into elementary changes as follows.

delta' = delta_10 composed with delta_9 composed with so forth down to delta_1, where
delta_1 is the change that inserts the character M at position 10
delta_2 is the change that inserts the character E at position 11
… and so on.

Note that what we consider an elementary change can depend on the context of the problem. We could consider insertions and deletions of lines in a file to be elementary. Or if we are using delta debugging on a set of binary parameters for a program, an elementary change might be switching one bit on or off.



Summary

- We have an input without failure: r_P
- We have an input with failure: r_F
- We have a set of changes $c_F = \{ \delta_1, \delta_2, \ldots, \delta_n \}$ such that:

$$r_F = (\delta_1 \circ \delta_2 \circ \ldots \circ \delta_n)(r_P)$$

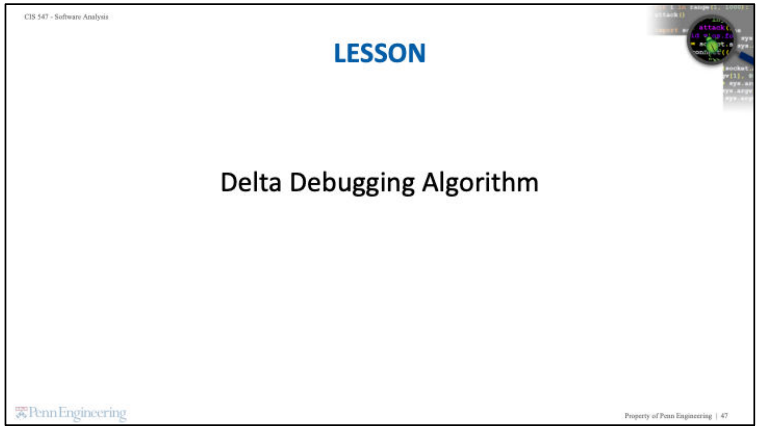
- Each subset c of c_F is a test case

Let's review the setup we have going into the delta debugging algorithm.

We have an input on which our program passes: r_p. We have an input on which our program fails: r_f. And we have a set of elementary changes, which we'll call c_f, such that applying the changes in order to r_p yields r_f.

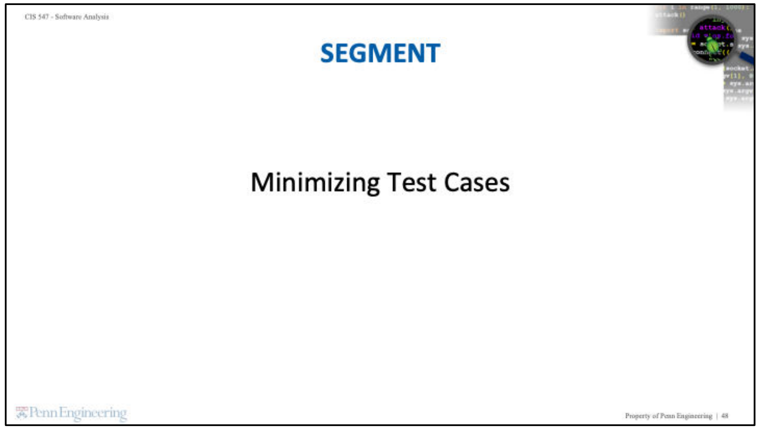
Note that r_p is typically a simple input on which the program trivially passes, such as the empty input, and r_f is typically a complex input on which the program fails, and that we would like to minimize. In the case of Mozilla web browser, r_p could be a blank HTML file, and r_f is the full HTML file that causes the browser to crash.

Subsets of c_f will be important in the delta debugging algorithm, so we will distinguish them henceforth by calling them test cases.



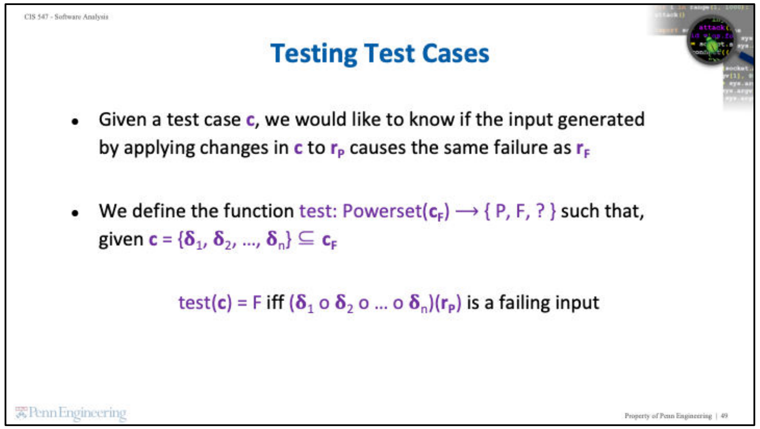
LESSON

Delta Debugging Algorithm



SEGMENT

Minimizing Test Cases



CIS 547 - Software Analysis

Testing Test Cases

- Given a test case **c**, we would like to know if the input generated by applying changes in **c** to r_P causes the same failure as r_F
- We define the function test: Powerset(c_F) ⟶ { P, F, ? } such that, given $c = \{\delta_1, \delta_2, \ldots, \delta_n\} \subseteq c_F$

$$\text{test}(c) = F \text{ iff } (\delta_1 \circ \delta_2 \circ \ldots \circ \delta_n)(r_P) \text{ is a failing input}$$

Penn Engineering

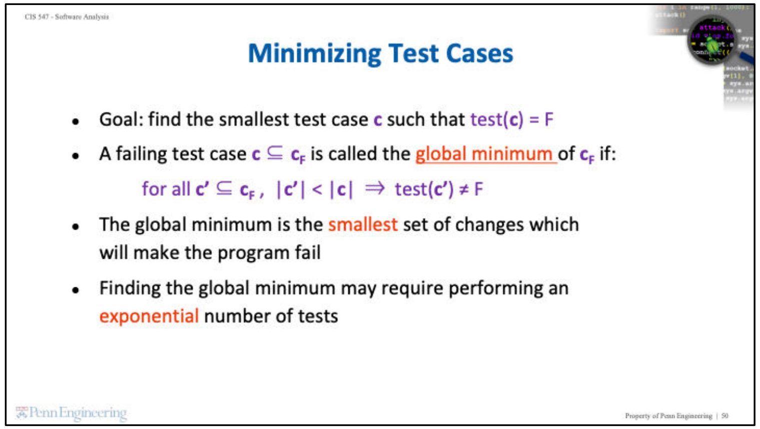
Property of Penn Engineering | 49

Given a test case, that is, a subset of the changes, we want the ability to apply that subset of changes to the passing input r_p and determine if the resulting input causes the program to fail in the same manner as the failing input r_f.

To formalize this notion, we define the function test which takes a subset of c_f and outputs one of three characters based on the outcome of our test.

We distinguish these three test outcomes following the POSIX 1003.3 standard for testing frameworks.

If the test succeeds, the test function outputs PASS, written here as P.
If the test produces the failure it was intended to capture, the test function outputs FAIL, written here as F.
And if the test produces indeterminate results, the test function outputs UNRESOLVED, written here as '?'.



CIS 547 - Software Analysis

Minimizing Test Cases

- Goal: find the smallest test case **c** such that test(**c**) = F
- A failing test case $c \subseteq c_F$ is called the global minimum of c_F if:

  for all $c' \subseteq c_F$, $|c'| < |c| \Rightarrow \text{test}(c') \neq F$
- The global minimum is the smallest set of changes which will make the program fail
- Finding the global minimum may require performing an exponential number of tests

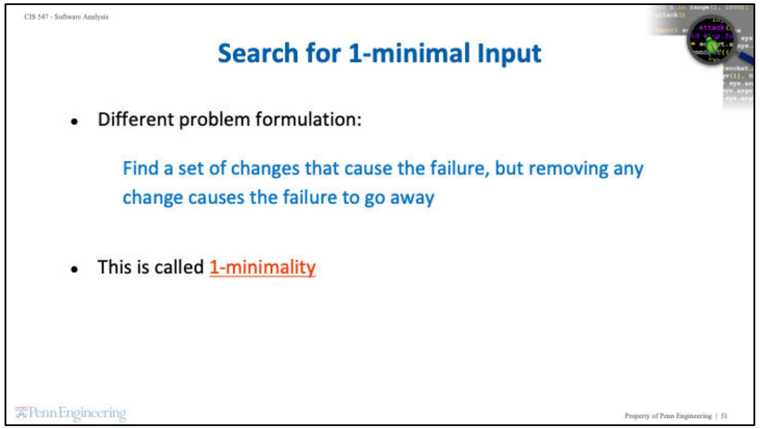
Penn Engineering

Property of Penn Engineering | 50

The goal of delta debugging is to find the smallest test case c such that test(c) = F. In other words, to find the smallest set of changes we need to apply to passing input r_p in order to result in the same failure as the failing input r_f.

We call a failing test case a global minimum of c_f if every other smaller-sized test case from c_f does not cause the test to output F. In other words, any smaller-sized set of changes either passes the test or causes the test to be unresolved.

The global minimum is the smallest set of changes which makes the program fail; but finding the global minimum may require performing an exponential number of tests: if c_f has size n, we'd need to perform in the worst case 2^n tests to find the global minimum.



CIS 547 - Software Analysis

Search for 1-minimal Input

- Different problem formulation:

  Find a set of changes that cause the failure, but removing any change causes the failure to go away

- This is called 1-minimality

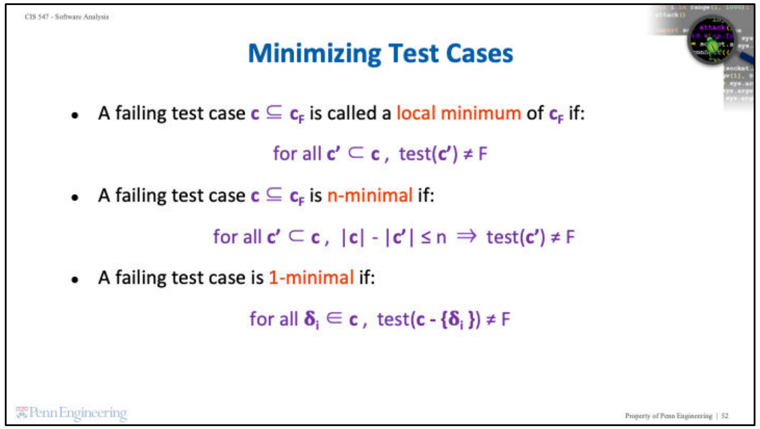
Penn Engineering

Property of Penn Engineering | 51

Instead of searching for the absolute smallest set of changes that causes the failure, we can approximate a smallest set by reformulating our goal.

We will find a set of changes that causes the failure but removing any single change from the set causes the failure to go away.

Such a set of changes is called 1-minimal.



CIS 547 - Software Analysis

Minimizing Test Cases

- A failing test case $\mathbf{c} \subseteq \mathbf{c_F}$ is called a local minimum of $\mathbf{c_F}$ if:

  $$\text{for all } \mathbf{c'} \subset \mathbf{c}, \ \text{test}(\mathbf{c'}) \neq \text{F}$$

- A failing test case $\mathbf{c} \subseteq \mathbf{c_F}$ is n-minimal if:

  $$\text{for all } \mathbf{c'} \subset \mathbf{c}, \ |\mathbf{c}| - |\mathbf{c'}| \leq n \Rightarrow \text{test}(\mathbf{c'}) \neq \text{F}$$

- A failing test case is 1-minimal if:

  $$\text{for all } \boldsymbol{\delta}_i \in \mathbf{c}, \ \text{test}(\mathbf{c} - \{\boldsymbol{\delta}_i\}) \neq \text{F}$$

Penn Engineering

Property of Penn Engineering | 52

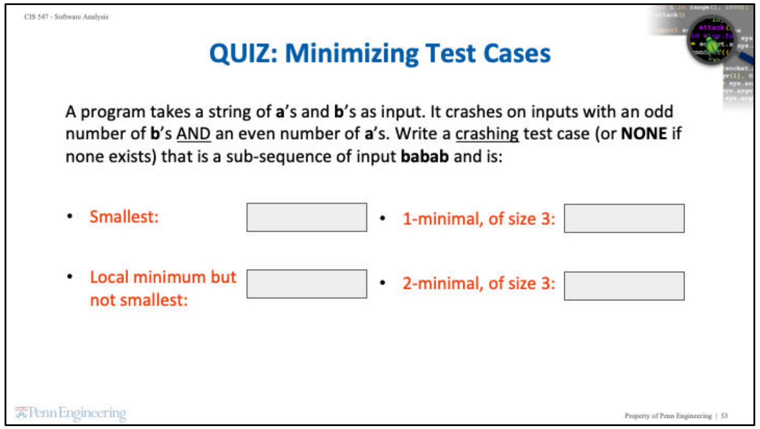
More formally:

Define a failing test case C to be a local minimum of C_F if for every proper subset C' of C, applying the test function to C' doesn't produce a failure. This is in contrast to a global minimum in the following way: for a local minimum, we only restrict our attention to subsets of the local minimum test case; for a global minimum, there must be no smaller test case that causes a failure.

Define a failing test case C to be n-minimal if for every proper subset C' of C, if the difference in size between C' and C is no more than n, then the test function applied to C' does not cause a failure. In other words, C is n-minimal if removing between 1 and n changes from C causes the test function to no longer fail. Just as local minimality is a weakening of the notion of global minimality, n-minimality is a weakening of the notion of local minimality.

And 1-minimality is the weakest form of n-minimality: a failing test case is 1-minimal if removing any single change from that test case causes the test function to no longer fail.

Even though 1-minimality is not nearly as strong as global or even local minimality, we focus on it because it is still a strong criterion: it says that the test case cannot be minimized incrementally. And we can program an efficient algorithm for applying and testing incremental changes.



CIS 547 - Software Analysis

QUIZ: Minimizing Test Cases

A program takes a string of **a**'s and **b**'s as input. It crashes on inputs with an odd number of **b**'s AND an even number of **a**'s. Write a crashing test case (or **NONE** if none exists) that is a sub-sequence of input **babab** and is:

- Smallest:
- Local minimum but not smallest:
- 1-minimal, of size 3:
- 2-minimal, of size 3:

Penn Engineering

Property of Penn Engineering | 53

{QUIZ SLIDE}

Let's stop here to check your understanding of the different types of minimality with a quiz.

Suppose a program takes a string of 'A's and 'B's as input, and it crashes if given an input with an odd number of 'B's AND an even number of 'A's.

Because 'BABAB' has an odd number of 'B's and an even number of A's, it is a failing input to the program. If we take r_p to be the empty input and r_f to be 'BABAB' and consider inserting each character to be a separate change, then c_f will be a set of five elementary changes.

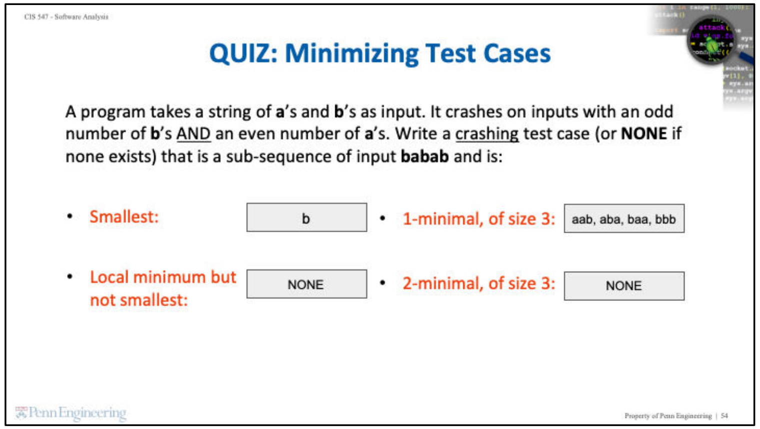
Previously we defined a test case to be a subset of these changes, which was a set of delta functions. For brevity, we won't use the delta notation in this quiz; instead, we'll slightly abuse terminology and just consider test cases to be the subsequences of 'BABAB' that result from applying those changes in a subset of c_f.

Here, I'd like you to enter four failure-inducing test cases that are subsequences of the input 'BABAB' satisfying the following constraints:

First, find the the global minimum test case (that is, the smallest possible failing subsequence)

Second, find a local minimum that is not the global minimum
Third, find a 1-minimal failing test case of size 3
And lastly, find a 2-minimal failing test case of size 3

If no subsequence of 'BABAB' exists satisfying the constraints, enter the word "NONE" instead.



CIS 547 - Software Analysis

QUIZ: Minimizing Test Cases

A program takes a string of **a**'s and **b**'s as input. It crashes on inputs with an odd number of **b**'s AND an even number of **a**'s. Write a crashing test case (or **NONE** if none exists) that is a sub-sequence of input **babab** and is:

- Smallest: b
- 1-minimal, of size 3: aab, aba, baa, bbb
- Local minimum but not smallest: NONE
- 2-minimal, of size 3: NONE

Penn Engineering

Property of Penn Engineering | 54

{SOLUTION SLIDE}

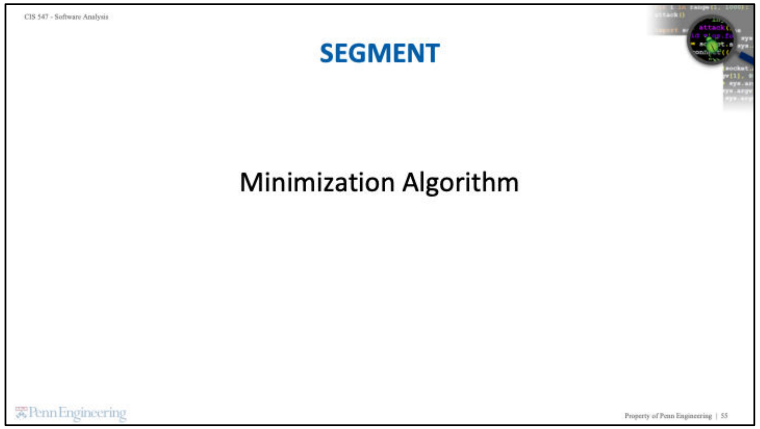
Let's start with the global minimum. Notice that the program crashes only on nonempty inputs (since we need to include at least one 'B' to have an odd number of 'B's), we start by considering subsequences of size 1. The only input of size 1 with at least one 'B' is the string consisting of just 'B'. This subsequence fails the test (it has 1 'B' -- an odd number, and 0 'A's -- an even number). Since 'B' is the smallest possible failing subsequence of 'BABAB', it is the global minimum failing test case.

Next, let's try to find a local minimum that is not a global minimum. Remember that no proper subsequence of a local minimum can fail. But earlier we said that all failing subsequences will need at least one 'B'. So every failing subsequence of 'BABAB' itself has a failing subsequence: 'B'. So the only local minimum is 'B' itself, so there are no local minima that are not global minima.

Now, let's try to find a 1-minimal failing subsequence of 'BABAB' of size 3. First, we'll list all failing subsequences of size 3: we need at least one 'B', and we need an even number of 'A's. This means we can either have 2 'A's and 1 'B' or 3 'B's. There are four subsequences of 'BABAB' that satisfy this criterion: 'AAB', 'ABA', 'BAA', and 'BBB'.

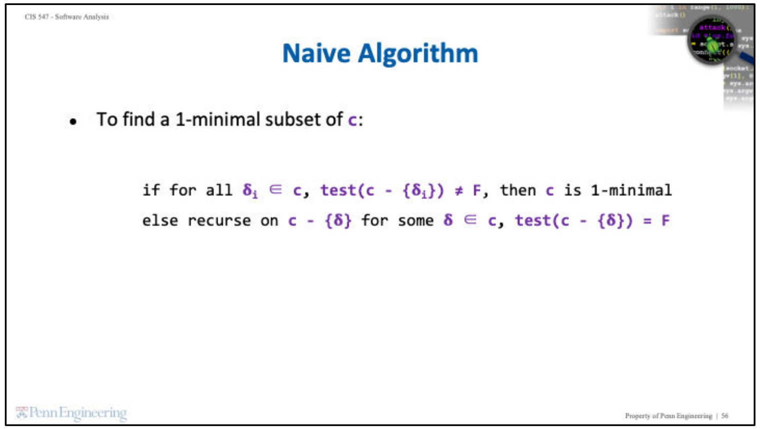
Now let's see which of these are 1-minimal. Remember that a failing test case is 1-minimal if, no matter which change we remove, we get a passing test case. Now, removing one character from any of these strings results in changing the parity of either the 'A's or the 'B's, meaning that the new subsequence will not cause a crash. Thus, all of these subsequences are 1-minimal.

Are any of them 2-minimal, however? This means that, in addition to removing one character, removing any two characters arbitrarily still causes the subsequence to pass. In this case, however, by removing two characters and leaving just a single 'B', we obtain a failing input. So none of these test cases are 2-minimal.



SEGMENT

Minimization Algorithm



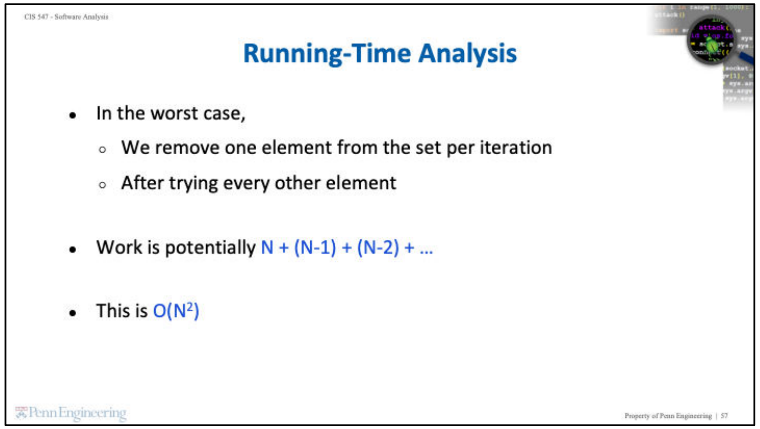
Naive Algorithm

- To find a 1-minimal subset of c:

```
if for all $\delta_i \in$ c, test(c - {$\delta_i$}) $\neq$ F, then c is 1-minimal
else recurse on c - {$\delta$} for some $\delta \in$ c, test(c - {$\delta$}) = F
```

Now, let’s think about how to build an algorithm to find a 1-minimal subset of a given set of changes c; we would then apply this algorithm to find the 1-minimal subset of the set of all changes by setting c to c_F.

One straightforward approach that might occur to us is to do the following: Iterate through each change delta_i in c, testing whether the set c minus delta_i fails or not. If we find a change delta such that c without delta still induces failure, then we call the algorithm recursively on c’ = c - {delta}. On the other hand, if every change’s removal causes the test to stop failing, then c is 1-minimal, so we return c.



CIS 547 - Software Analysis

Running-Time Analysis

- In the worst case,
  - We remove one element from the set per iteration
  - After trying every other element
- Work is potentially N + (N-1) + (N-2) + …
- This is $O(N^2)$

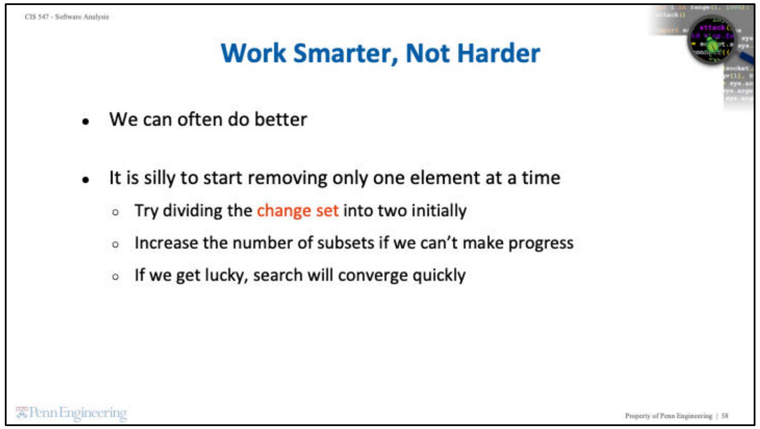
Penn Engineering

Property of Penn Engineering | 57

How well does this naive approach work? Well, in the worst case, we would remove the last change in the list per iteration after testing all previous changes.

If we start with N elements, then we perform up to N-i tests on the ith iteration (starting from iteration 0). The total number of tests in the worst case would then be N plus N-1 plus N-2 and so forth.

For large values of N, this is approximately one-half N^2, or O(N^2) in asymptotic notation.



CIS 547 - Software Analysis

Work Smarter, Not Harder

- We can often do better
- It is silly to start removing only one element at a time
  - Try dividing the change set into two initially
  - Increase the number of subsets if we can't make progress
  - If we get lucky, search will converge quickly

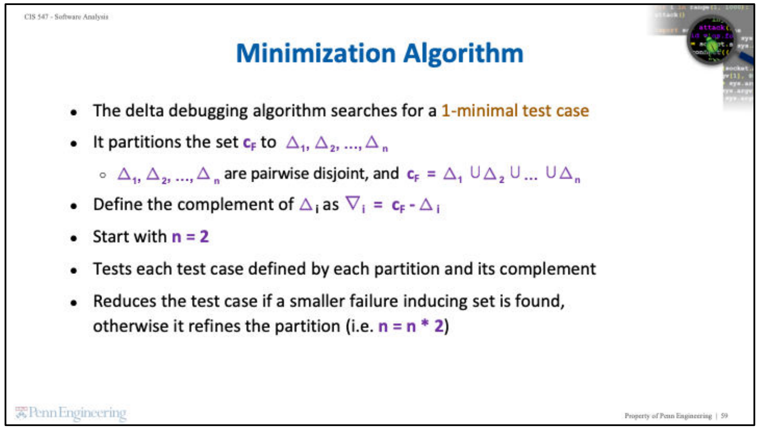
Penn Engineering

Property of Penn Engineering | 58

We can often attain better performance than the first, simplest algorithm we thought of. Let's try to see if we can improve our algorithm's performance by making some modifications.

What's one place where we are losing time in our algorithm?

Recall our discussion earlier about the strengths and weaknesses of coarser versus finer granularity. Checking one change at a time is very fine granularity, which allows for a greater chance of success in finding a failure-inducing subset of changes. But it is also more time-consuming. If we start with very coarse changes at first, we might be able to save a lot of time; only if we can't make any progress should we refine our granularity and increase the number of subsets we test.



Minimization Algorithm

- The delta debugging algorithm searches for a 1-minimal test case
- It partitions the set c_F to $\triangle_1, \triangle_2, ..., \triangle_n$
  - $\triangle_1, \triangle_2, ..., \triangle_n$ are pairwise disjoint, and $c_F = \triangle_1 \cup \triangle_2 \cup ... \cup \triangle_n$
- Define the complement of $\triangle_i$ as $\nabla_i = c_F - \triangle_i$
- Start with $n = 2$
- Tests each test case defined by each partition and its complement
- Reduces the test case if a smaller failure inducing set is found, otherwise it refines the partition (i.e. $n = n * 2$)

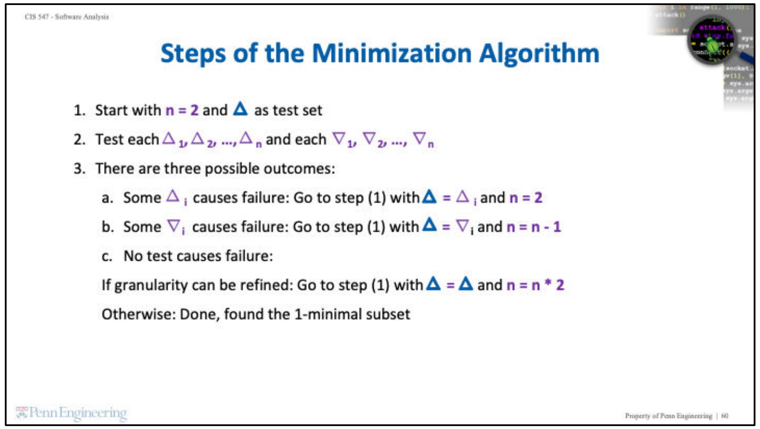
Here is a sketch of the delta debugging minimization algorithm invented by Andreas Zeller who originally proposed delta debugging. The algorithm finds a 1-minimal test case from the given set of changes c_F.

It starts with n = 2, and divides the set c_F up into n pairwise-disjoint pieces, called Delta_1 through Delta_N. (We use capital deltas here to represent subsets of c_F instead of individual changes in c_F.)

We also use Nabla, the upside-down capital Delta, to represent the complement in c_F of each capital Delta. In other words, all the changes in c_F which aren't in Delta_i are in Nabla_i.

The algorithm then applies the test function to each Delta_i and each Nabla_i.

If one of these test cases fails, then the algorithm reduces the current input down to the input obtained by just applying the changes in the failing test case. If none of the test cases fails, though, then the algorithm refines its granularity by doubling n and recomputing new subsets Delta_i and Nabla_i.



Steps of the Minimization Algorithm

1. Start with $n = 2$ and Δ as test set
2. Test each $\triangle_1, \triangle_2, ..., \triangle_n$ and each $\nabla_1, \nabla_2, ..., \nabla_n$
3. There are three possible outcomes:
   a. Some $\triangle_i$ causes failure: Go to step (1) with $\Delta = \triangle_i$ and $n = 2$
   b. Some ∇_i causes failure: Go to step (1) with $\Delta = \nabla_i$ and $n = n - 1$
   c. No test causes failure:
   If granularity can be refined: Go to step (1) with $\Delta = \Delta$ and $n = n * 2$
   Otherwise: Done, found the 1-minimal subset

Here is the algorithm again, this time in more structured pseudocode. It has two parameters n and Delta.

It starts with n = 2 and Delta equal to c_F, the full set of elementary changes.

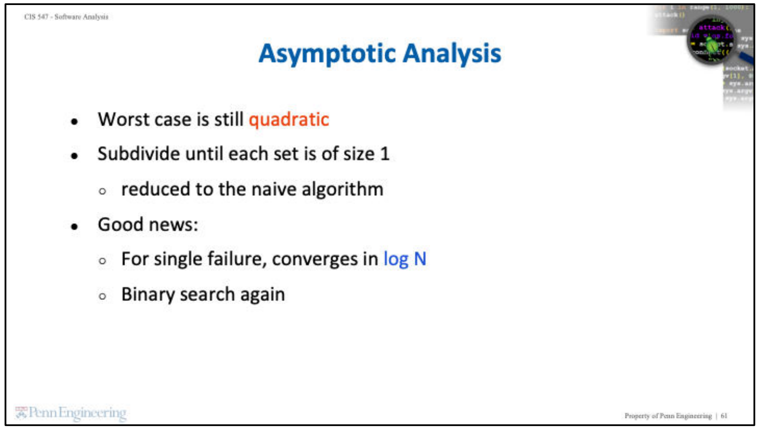
Given n and Delta, the algorithm divides Delta up into n pieces, Delta_1 through Delta_n, and computes Nabla_1 through Nabla_n appropriately.

It then tests each Delta_i and Nabla_i using the test function. There are three possible outcomes:

If some Delta_i causes the test function to fail, then we go back to step (1), this time with Delta = Delta_i and resetting n to 2.

Otherwise, if some Nabla_i causes the test function to fail, then we go back to step (1), this time with Delta = Nabla_i and decrementing n by 1.

If none of the test cases causes a failure, then we have two possibilities:
If the granularity is not yet at its maximum (n < size of Delta), we return to step (1), leaving Delta the same and doubling the granularity.
If the granularity is already at maximum (n >= size of Delta), then this means each capital Delta_i consists of a single change, and removing any single change causes the test case to no longer fail. Thus, the test case is 1-minimal, and we stop.



Asymptotic Analysis

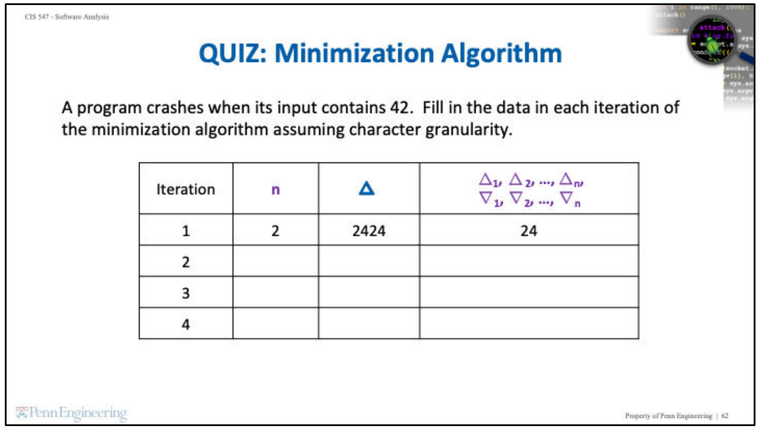
- Worst case is still quadratic
- Subdivide until each set is of size 1
  - reduced to the naive algorithm
- Good news:
  - For single failure, converges in log N
  - Binary search again

Let's analyze this algorithm's complexity and see how it compares to our previous attempt.

Unfortunately, the worst-case complexity of delta debugging minimization is still quadratic in the number of elementary changes: it could be the case that we need to subdivide until we reach maximum granularity and then we remove one change at a time, effectively doing the same amount of work as the naive algorithm.

(As an exercise for yourself, try to come up with an example of a test function and family of inputs that would give this worst case scenario!)

The good news is that in the case where we find a failure in either Delta_1 or Delta_2 in each iteration, convergence to the 1-minimal test case takes only a logarithmic number of tests (much like binary search).



QUIZ: Minimization Algorithm

A program crashes when its input contains 42. Fill in the data in each iteration of the minimization algorithm assuming character granularity.

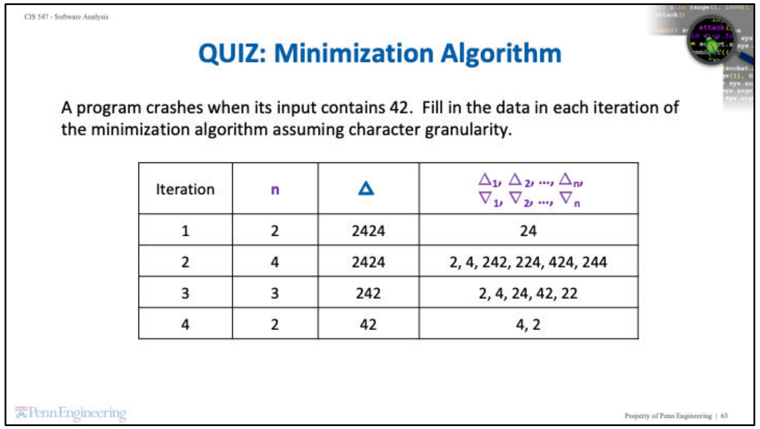
Iteration	n	Δ	$\Delta_1, \Delta_2, \ldots, \Delta_n,$ $\nabla_1, \nabla_2, \ldots, \nabla_n$
1	2	2424	24
2			
3			
4			

{QUIZ SLIDE}

Let's work through an example of the minimization algorithm in the form of a quiz. Suppose a program crashes whenever its input contains the substring '42', and suppose we start with the original failing string '2424'. Assuming that each elementary change consists of inserting a single character, let's see how the algorithm would minimize this string.

First, begin by filling in the number of partitions we would make of the string Delta, and write in the strings that would form our test cases. Please separate the strings by commas, and don't surround the strings by quotation marks. Also feel free to ignore duplicate strings (for example, if both Delta_1 and Delta_2 are the same in some iteration, you just need to write it once).

Finally, if Delta cannot be partitioned evenly into n groups, split it into groups as evenly as possible.



CIS 547 - Software Analysis

QUIZ: Minimization Algorithm

A program crashes when its input contains 42. Fill in the data in each iteration of the minimization algorithm assuming character granularity.

Iteration	n	Δ	$\Delta_1, \Delta_2, \ldots, \Delta_n, \nabla_1, \nabla_2, \ldots, \nabla_n$
1	2	2424	24
2	4	2424	2, 4, 242, 224, 424, 244
3	3	242	2, 4, 24, 42, 22
4	2	42	4, 2

Penn Engineering

Property of Penn Engineering | 63

{SOLUTION SLIDE}

In the first iteration, we start with n = 2, and dividing Delta = 2424 into two even groups gives the same string for all of Delta_1, Delta_2, Nabla_1, and Nabla_2: 24. Since 24 does not cause the program to fail, we leave Delta the same for iteration 2 but double the number of partitions to 4.

Dividing up 2424 into four partitions yields
Delta_1 = Delta_3 = 2
Delta_2 = Delta_4 = 4
Nabla_1 = 424
Nabla_2 = 224
Nabla_3 = 244, and
Nabla_4 = 242.

Nabla_1 and Nabla_4 are the only ones that fail, so we may choose either of them as Delta and proceed. Either way, we would decrement n by 1 to get n = 3.

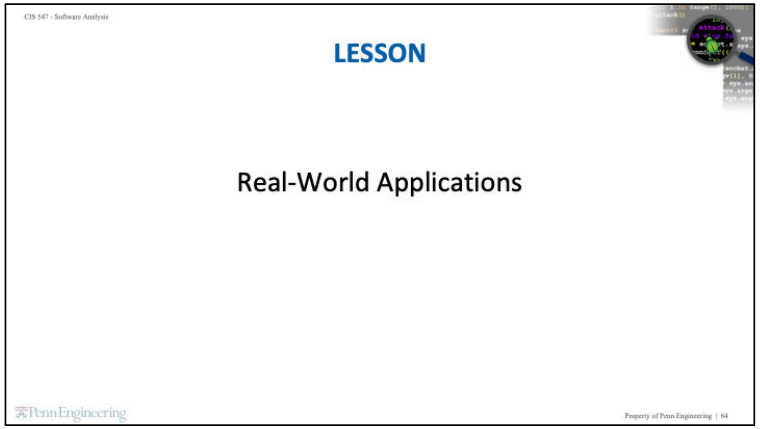
If we picked Nabla_4 = 242, then our partition would yield
Delta_1 = Delta_3 = 2
Delta_2 = 4
Nabla_1 = 42
Nabla_2 = 22, and
Nabla_3 = 24.

(If we had picked Nabla_1 = 424 earlier in iteration 2, then our partition would yield the same set except that 22 would be replaced by 44.)

Either way, the only failing test case would be 42, which we take Delta to be. We also decrement n to 2.

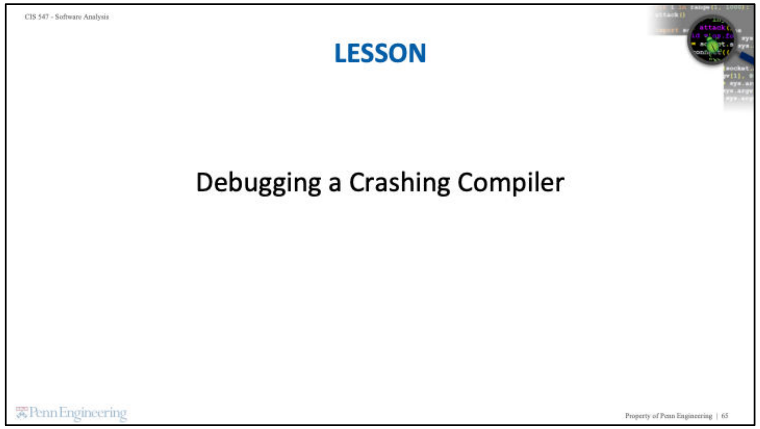
Finally, partitioning 42 into two parts gives Delta_1 = Nabla_2 = 4 and Delta_2 = Nabla_1 = 2.

None of these test cases fails, and we observe that n equals the size of Delta. Thus, our algorithm terminates and returns Delta = 42 as the minimized failing test case.



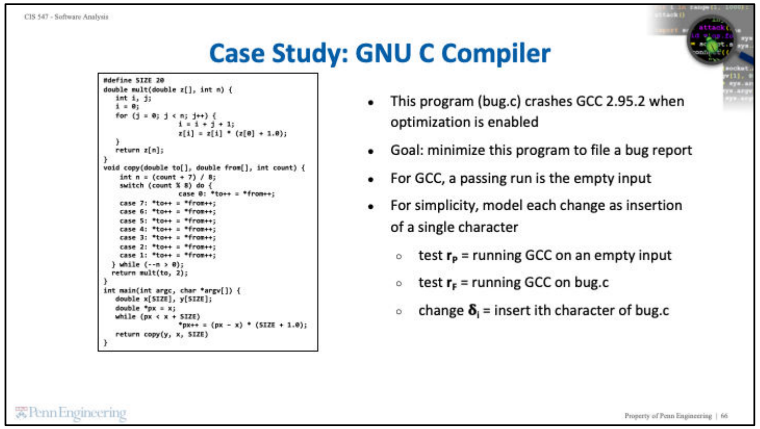
LESSON

Real-World Applications



LESSON

Debugging a Crashing Compiler



Case Study: GNU C Compiler

```
#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)
}
```

- This program (bug.c) crashes GCC 2.95.2 when optimization is enabled
- Goal: minimize this program to file a bug report
- For GCC, a passing run is the empty input
- For simplicity, model each change as insertion of a single character
  - ○ test $\mathbf{r_P}$ = running GCC on an empty input
  - ○ test $\mathbf{r_F}$ = running GCC on bug.c
  - ○ change $\boldsymbol{\delta}_i$ = insert ith character of bug.c

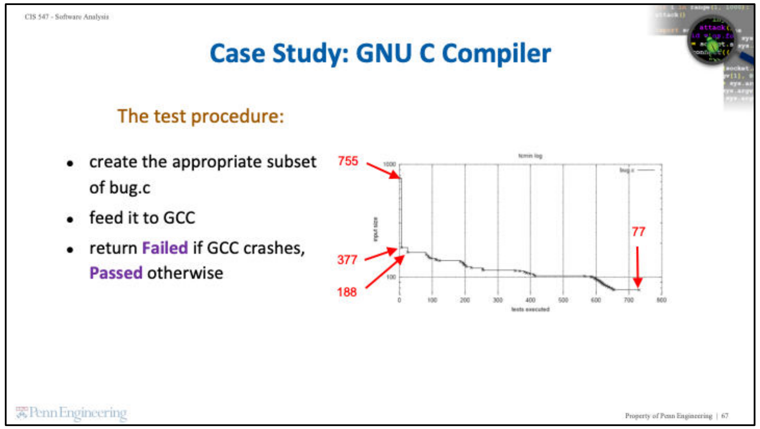
In the rest of this module, I will illustrate the versatility of delta debugging using a series of case studies conducted by the author of the technique.

You can learn more about these case studies as well as the delta debugging technique by following this link to a technical paper:

https://www.st.cs.uni-saarland.de/publications/files/zeller-tse-2002.pdf

The following C program, denoted bug.c, causes GCC version 2.95.2 with optimizations enabled to crash. This program consists of three functions: mult, copy, and main. Suppose we wish to minimize the program to file a bug report on GCC.

Delta debugging can be used to achieve this goal. For the GCC program, a passing input is the empty input. And, for the sake of simplicity, let’s model each change as an insertion of a single character. Then, in the terminology of the delta debugging algorithm, test r_p denotes running GCC on an empty input, test r_f denotes running GCC on bug.c, which is this entire input, and each change delta_i denotes inserting the i_th character of bug.c.



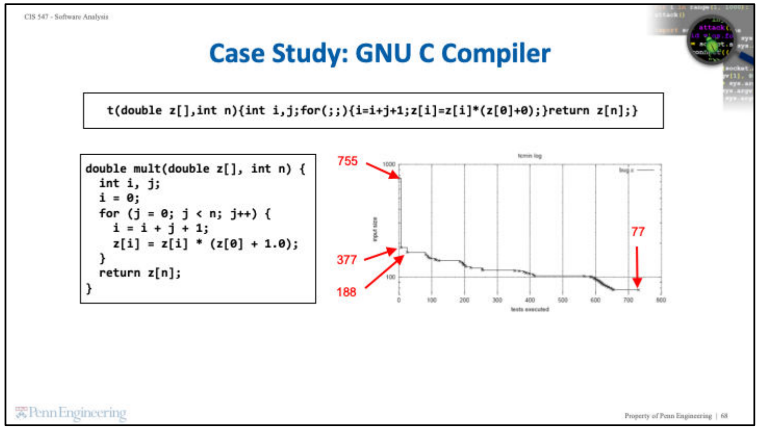
Case Study: GNU C Compiler

The test procedure:

- create the appropriate subset of bug.c
- feed it to GCC
- return **Failed** if GCC crashes, **Passed** otherwise

We next write the test procedure to be provided to the delta debugging algorithm. This procedure consists of three steps. First, it creates the appropriate subset of bug.c. Next, it feeds this subset to GCC. Finally, it returns Failed if GCC crashes, and Passed otherwise.

We then run the delta debugging algorithm using this test procedure. In only the first two tests, the algorithm reduces the input size from 755 characters to 377 and 188 characters, respectively.

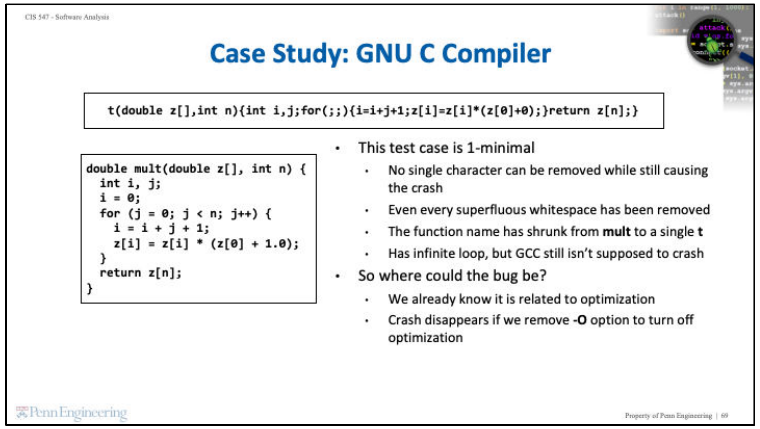


Case Study: GNU C Compiler

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

```
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];
}
```

The test case now only contains the mult function: the copy and main functions have been eliminated. Reducing mult, however, takes time. Only after 731 more tests do we get a test case that cannot be minimized further. This test case only contains 77 characters.



Case Study: GNU C Compiler

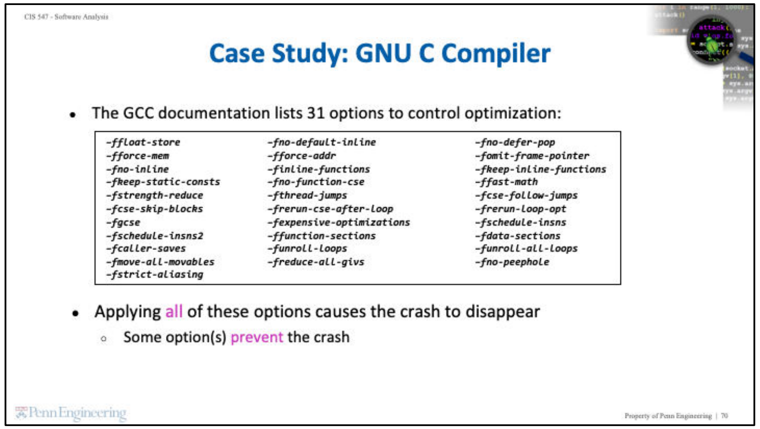
```
t(double z[],int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*(z[0]+0);}return z[n];}
```

```
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];
}
```

- This test case is 1-minimal
  - No single character can be removed while still causing the crash
  - Even every superfluous whitespace has been removed
  - The function name has shrunk from **mult** to a single **t**
  - Has infinite loop, but GCC still isn't supposed to crash
- So where could the bug be?
  - We already know it is related to optimization
  - Crash disappears if we remove **-O** option to turn off optimization

This test case is 1-minimal, because no single character can be removed while still causing GCC to crash. Notice how every superfluous whitespace has been removed. Even the function name has shrunk from mult to a single letter t, and the original loop has been converted to an infinite loop. But GCC still isn't supposed to crash.

As GCC users, we can now file this one-line program as a minimal bug report. But where in the GCC code could the bug be? We already know it is related to GCC optimization: the crash disappears if we remove the -O option on the command line to turn off optimization.



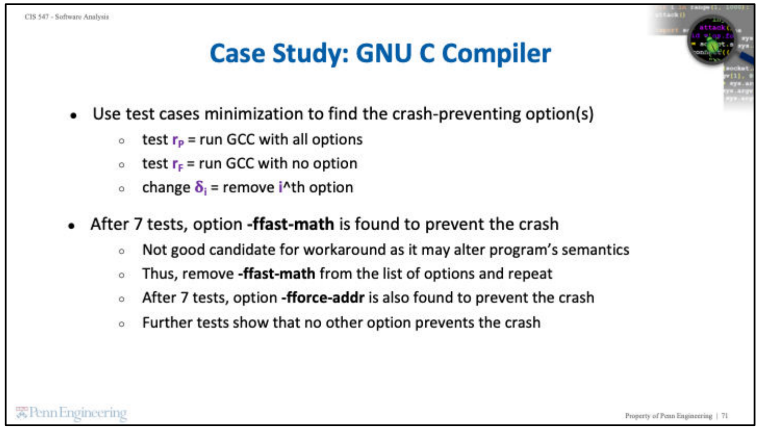
Case Study: GNU C Compiler

- The GCC documentation lists 31 options to control optimization:

```
-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
```

- Applying all of these options causes the crash to disappear
  - Some option(s) prevent the crash

Now, the GCC documentation lists 31 different options to control optimization. It turns out that applying all of these options causes the crash to disappear. This means that some options in this list prevent the crash.



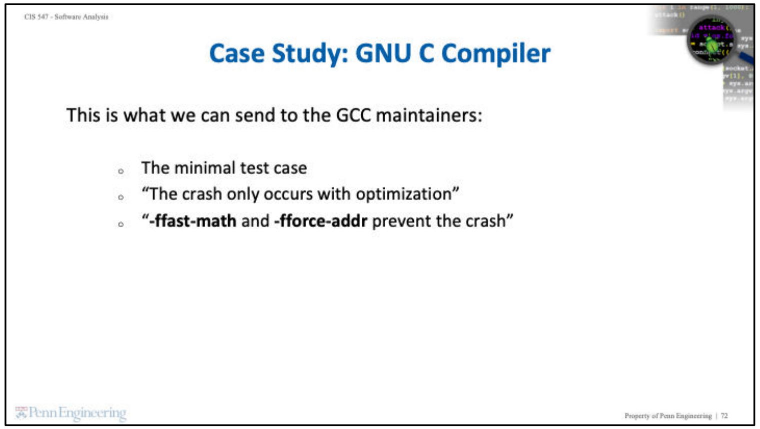
Case Study: GNU C Compiler

- Use test cases minimization to find the crash-preventing option(s)
  - test r_P = run GCC with all options
  - test r_F = run GCC with no option
  - change δ_i = remove i^th option
- After 7 tests, option **-ffast-math** is found to prevent the crash
  - Not good candidate for workaround as it may alter program's semantics
  - Thus, remove **-ffast-math** from the list of options and repeat
  - After 7 tests, option **-fforce-addr** is also found to prevent the crash
  - Further tests show that no other option prevents the crash

We can again use the delta debugging algorithm to find the crash-preventing options. This time, the passing test r_p denotes running GCC with all options, the failing test r_f denotes running GCC with none of the options, and each change delta_i denotes removing the i^th option.

After 7 tests, the algorithm reports that option -ffast-math prevents the crash. Unfortunately, the –ffast-math option is a bad candidate for working around the failure, because it may alter the semantics of the program. So we remove –ffast-math from the list of options and re-run the delta debugging algorithm. Again after 7 tests, it turns out the option –fforce-addr also prevents the crash.

So far, we have determined that 2 of the 31 options prevent the crash. Running GCC with the remaining 29 options shows that the crash persists; so it seems we have identified all the crash-preventing options.



CIS 547 - Software Analysis

Case Study: GNU C Compiler

This is what we can send to the GCC maintainers:

- The minimal test case
- “The crash only occurs with optimization”
- “**-ffast-math** and **-fforce-addr** prevent the crash”

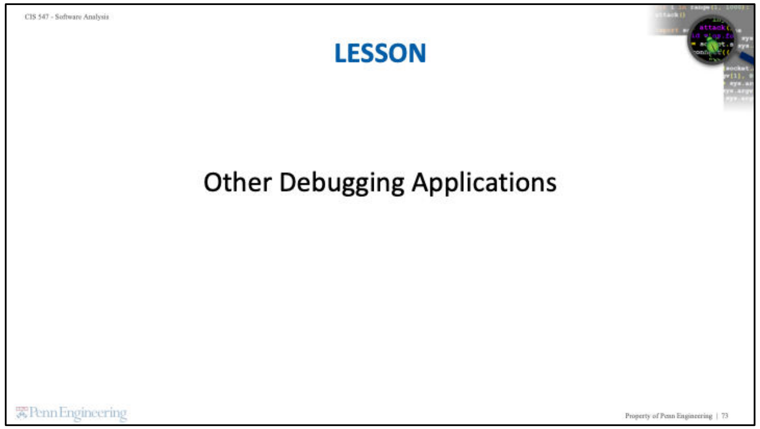
Penn Engineering

Property of Penn Engineering | 72

So this is what we can send to the GCC maintainers:

1. The minimal test case
2. The fact that “The crash occurs only with optimization”
3. and the fact that optimization options “–ffast-math and –fforce-addr prevent the crash.”

While we as GCC users cannot identify a place in the GCC code that causes the problem, we have identified as many failure circumstances as we can.



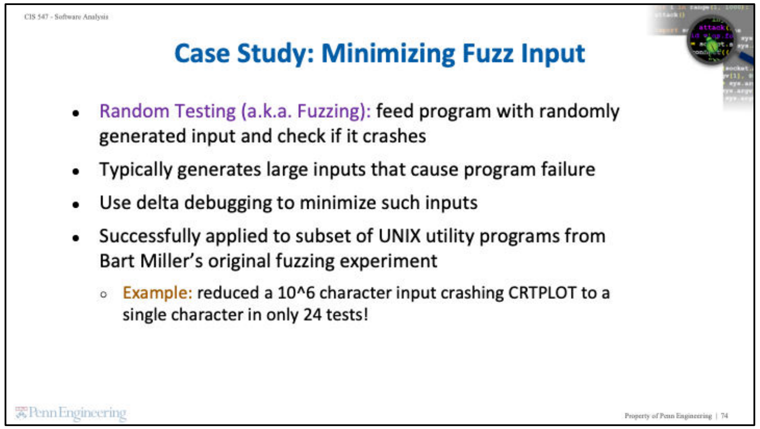
CIS 547 - Software Analysis

LESSON

Other Debugging Applications

Penn Engineering

Property of Penn Engineering | 73



CIS 547 - Software Analysis

Case Study: Minimizing Fuzz Input

- Random Testing (a.k.a. Fuzzing): feed program with randomly generated input and check if it crashes
- Typically generates large inputs that cause program failure
- Use delta debugging to minimize such inputs
- Successfully applied to subset of UNIX utility programs from Bart Miller's original fuzzing experiment
  - Example: reduced a 10^6 character input crashing CRTPLOT to a single character in only 24 tests!

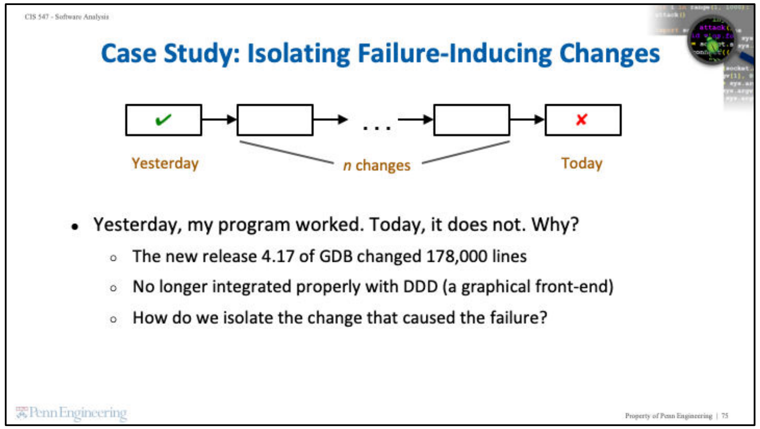
Penn Engineering

Property of Penn Engineering | 74

Another application of delta debugging is in the minimization of fuzz input, in which a program is fed with randomly generated inputs and observed to see if it crashes. Typically the failure-inducing inputs found by fuzzing are large; delta debugging can be used to reduce such inputs down to smaller inputs causing the same mode of failure.

Recall from the module on random testing that Bart Miller and his team examined the robustness of UNIX utilities by feeding them fuzz input -- a large number of random characters. The studies showed that 40% of these programs crash when fed with fuzz input.

The author of delta debugging successfully applied the technique to minimize the fuzz inputs that crash a subset of the UNIX utility programs. For example, the technique only required 24 tests to minimize a fuzz input comprising a 10^6 characters that crashes CRTPLOT to a single character that still crashes CRTPLOT in the same manner.



CIS 547 - Software Analysis

Case Study: Isolating Failure-Inducing Changes

✓
. . .
✗
Yesterday
n changes
Today

- Yesterday, my program worked. Today, it does not. Why?
  - The new release 4.17 of GDB changed 178,000 lines
  - No longer integrated properly with DDD (a graphical front-end)
  - How do we isolate the change that caused the failure?

Penn Engineering

Property of Penn Engineering | 75

Yet another application of delta debugging is to isolate changes to source code that cause program failure.

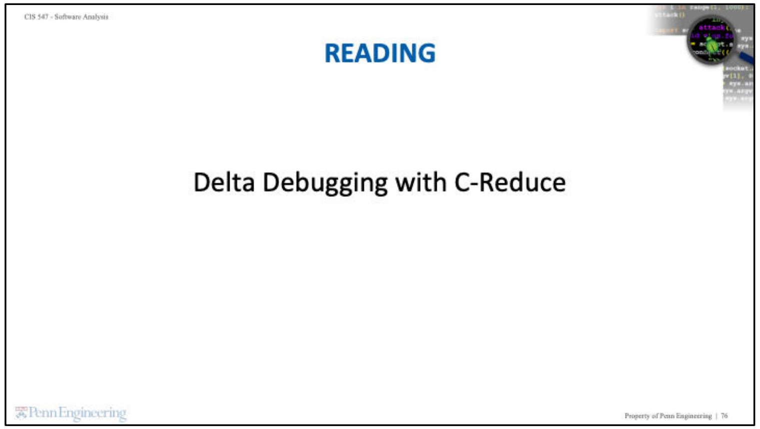
You likely have had this experience: one day, your program works fine; the next day, it does not, and you need to figure out why.

Perhaps the amount of code that's changed is quite large. For example, a certain release of GDB (the GNU debugger on UNIX) changed 178,000 lines. After this release, GDB no longer integrated correctly with the Data Display Debugger (or DDD), a common graphical user interface for GDB. How should the GDB maintainers determine which changed line (or lines) among those 178,000 lines is the culprit?

The solution is to use the delta debugging minimization algorithm with the passing input r_p being "yesterday's code" and the failing input r_f being "today's code." This allows you to pinpoint what specific change is making the code to no longer work.

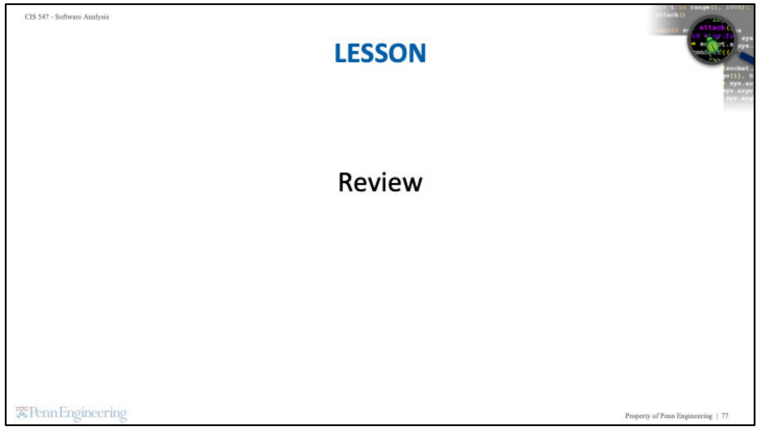
Further reading on this topic can be found at:

https://www.st.cs.uni-saarland.de/publications/files/zeller-esec-1999.pdf



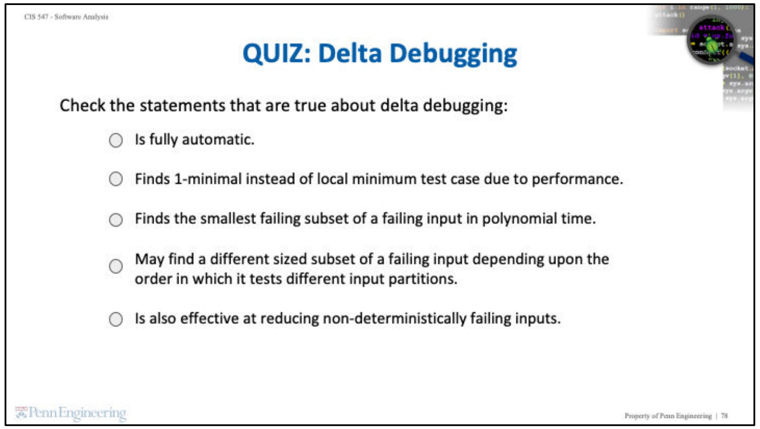
READING

Delta Debugging with C-Reduce



LESSON

Review



CIS 547 - Software Analysis

QUIZ: Delta Debugging

Check the statements that are true about delta debugging:

○ Is fully automatic.

○ Finds 1-minimal instead of local minimum test case due to performance.

○ Finds the smallest failing subset of a failing input in polynomial time.

○ May find a different sized subset of a failing input depending upon the order in which it tests different input partitions.

○ Is also effective at reducing non-deterministically failing inputs.

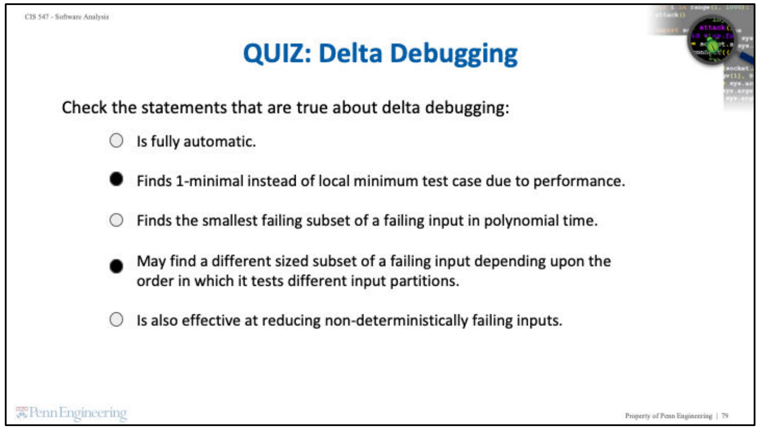
Penn Engineering

Property of Penn Engineering | 78

{QUIZ SLIDE}

As we close this lesson, let's recap the key concepts with the following quiz. Check all the statements that are true about delta debugging.

The technique is fully automatic.
It finds a 1-minimal test case instead of a local minimum test case due to performance reasons.
It finds the smallest failing subset of a failing input in polynomial time.
It may find a different sized subset of a failing input depending on the order in which it tests different input partitions.
It is also effective at reducing nondeterministically failing inputs.



CIS 547 - Software Analysis

QUIZ: Delta Debugging

Check the statements that are true about delta debugging:

○ Is fully automatic.

● Finds 1-minimal instead of local minimum test case due to performance.

○ Finds the smallest failing subset of a failing input in polynomial time.

● May find a different sized subset of a failing input depending upon the order in which it tests different input partitions.

○ Is also effective at reducing non-deterministically failing inputs.

Penn Engineering

Property of Penn Engineering | 79

{SOLUTION SLIDE}

Let's tackle each of the statements in order.

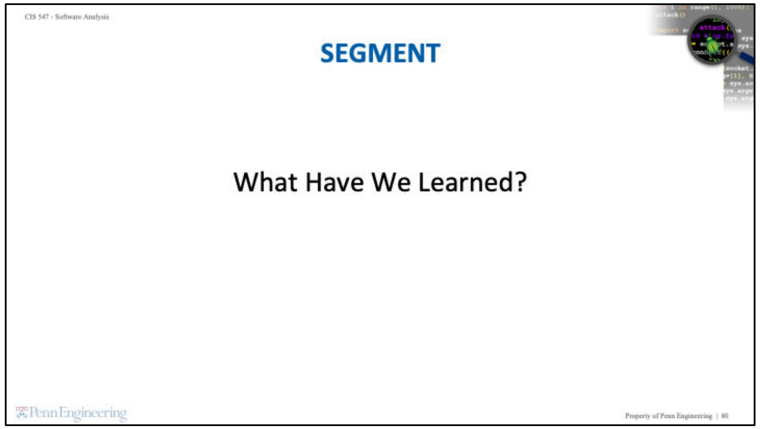
The technique is fully automatic. This is false because one has to define the space of input changes, or deltas, which is application-specific, as well as what constitutes a passing versus failing program run under each possible input.

Delta debugging finds a 1-minimal test case instead of a local minimum test case due to performance reasons. This is true: finding a local minimum (in the worst case) can also take exponential time in the number of changes. Finding a 1-minimal test case, however, takes at worst quadratic time.

Delta debugging finds the smallest failing subset of a failing input in polynomial time. This is false. The algorithm does not find the smallest failing subset: such a subset is the global minimum, which takes exponential time in the number of changes to find.

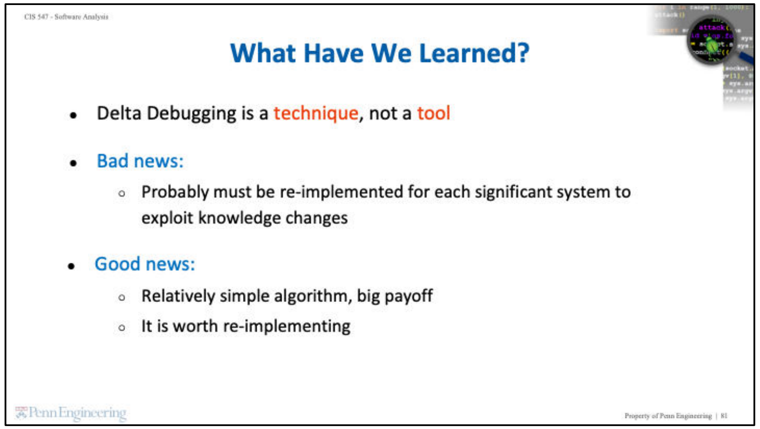
Delta debugging may find a different sized subset of a failing input depending on the order in which it tests different input partitions. This is also true, and here's a simple example to illustrate why. Consider a program that fails if its input contains either 'a' or 'bb'. The input 'aabb' therefore crashes. If the minimization algorithm examines 'aa' before 'bb' on the first iteration, then it will end up with the 1-minimal test case 'a'; on the other hand, if it examines 'bb' before 'aa' on the first iteration, it will end up with the 1-minimal test case ‘bb’.

Delta debugging is also effective at reducing non-deterministically failing inputs. This is false. The algorithm only functions correctly assuming that program failure is deterministic.



SEGMENT

What Have We Learned?



CIS 547 - Software Analysis

What Have We Learned?

- Delta Debugging is a technique, not a tool
- Bad news:
  - Probably must be re-implemented for each significant system to exploit knowledge changes
- Good news:
  - Relatively simple algorithm, big payoff
  - It is worth re-implementing

Penn Engineering

Property of Penn Engineering | 81

Let’s conclude by reviewing what we have learned about delta debugging in this module.

First of all, delta debugging, like random testing, is a technique, as opposed to a tool that can be used out-of-the-box. A limitation of the technique is that it is not readily portable across programs: it needs to be re-implemented for each significant system in order to exploit knowledge changes that are specific to the system. For example, a delta-debugging implementation for testing whether Mozilla’s browser crashes differs from one for testing optimization flags for the GCC compiler: these two scenarios have different notions of what constitutes an elementary change (perhaps a line or a character is an elementary change for the browser while a binary flag is an elementary change for the compiler).

The good news is that the delta debugging algorithm is relatively simple and provides excellent payoff for the effort it takes to implement it; therefore, it is worth re-implementing it across several systems.