Testing în practice

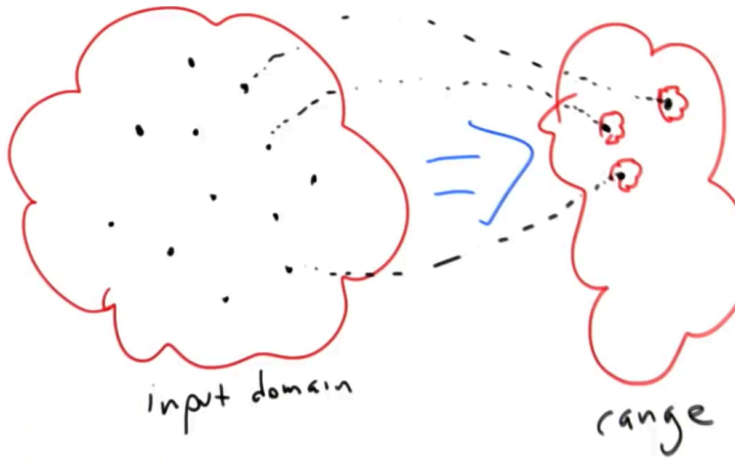
https://youtube.com/playlist?list=PLAwxTw4SYaPkWVHeC_8aSIbSxE_NXI76g&si=lhs0mKJKm5pW3D1R

1. Introduction

- In the following we discuss advanced testing issues, such as
  - **what to do when a false tester overloads us with bugs,**
  - **how to take a large test case that makes software fail and turn it into a very small test case,**
  - **how to report bugs in such a way the developers are more likely to pay attention to them**.

2. Overwhelmed by success

- One issue that can come up if we write a really good random test-case generator is that we can be overwhelmed by the success of our own bug-finding effort.
- Suppose we test a large software project using a sophisticated random tester. There might be 250 bugs.

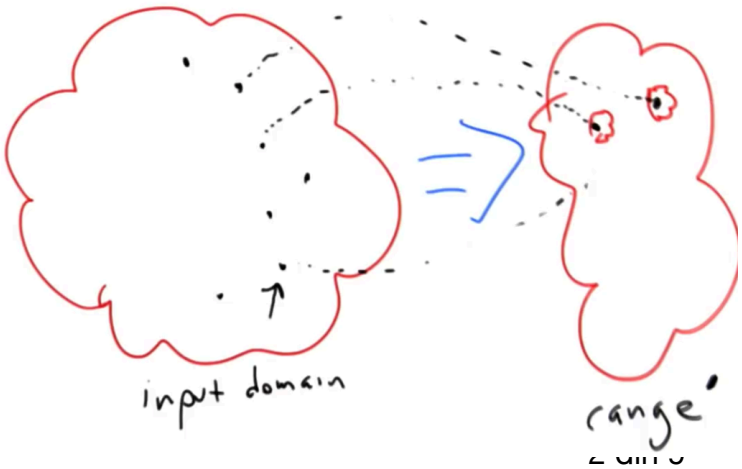


•Almost certainly we found 10 bugs or maybe 50 bugs, but we probably didn't get so lucky as to find 250 bugs.

•We probably weren't so unlucky as to trigger the same bug 250 times.

•There are 2 ways out of this problem.

- Pick a bug and report it to the developers. They're going to fix it:

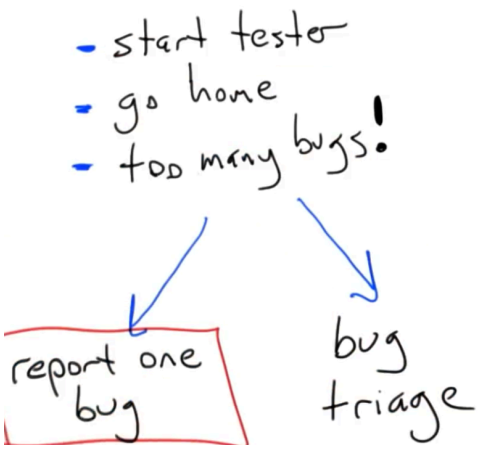


• So, we still have exactly the same input space but the behavior of the SUT has changed a little bit.

•Now something interesting happens: not only did that particular failure go away, but perhaps some of the other ones did. Perhaps all stopped being

inputs that trigger failures anymore.

- Another possibility is all of the remaining bug-triggering inputs that we found still trigger bugs, and so we report another bug. This strategy will keep working.

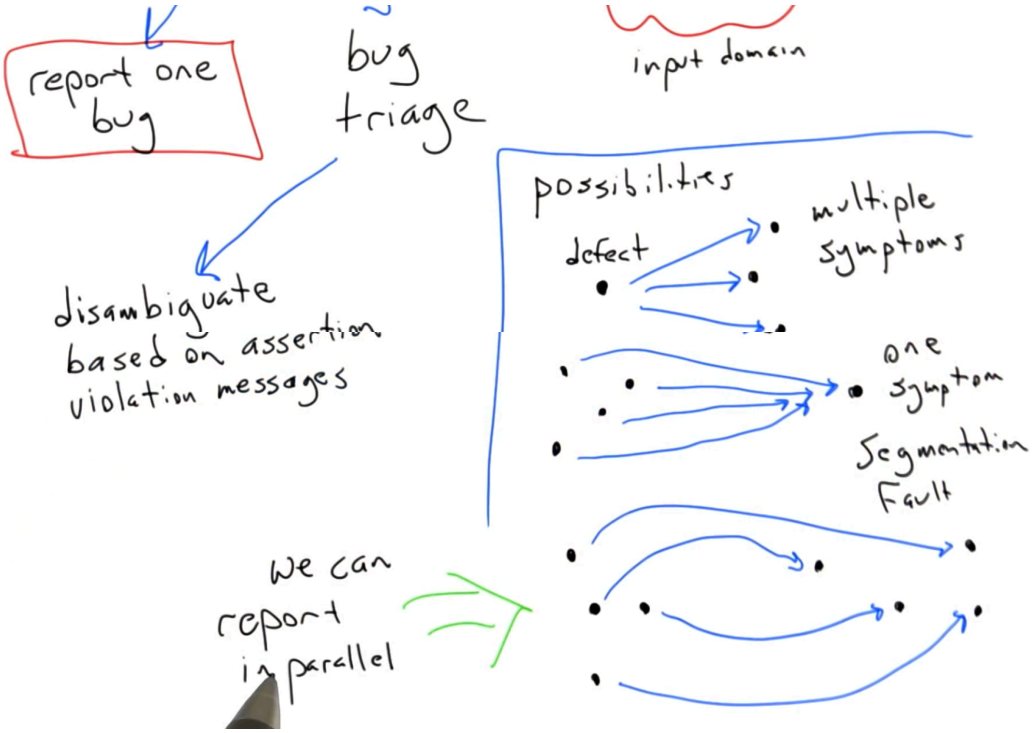


•As soon as we get a new version of the system that fixes a bug we've reported, we can just do another one. This isn't a bad mode of operation, but it works on small systems where bugs can be fixed rather quickly. For the complex ones it might take years to get a new version using a **one-bug-at-a-time model**.

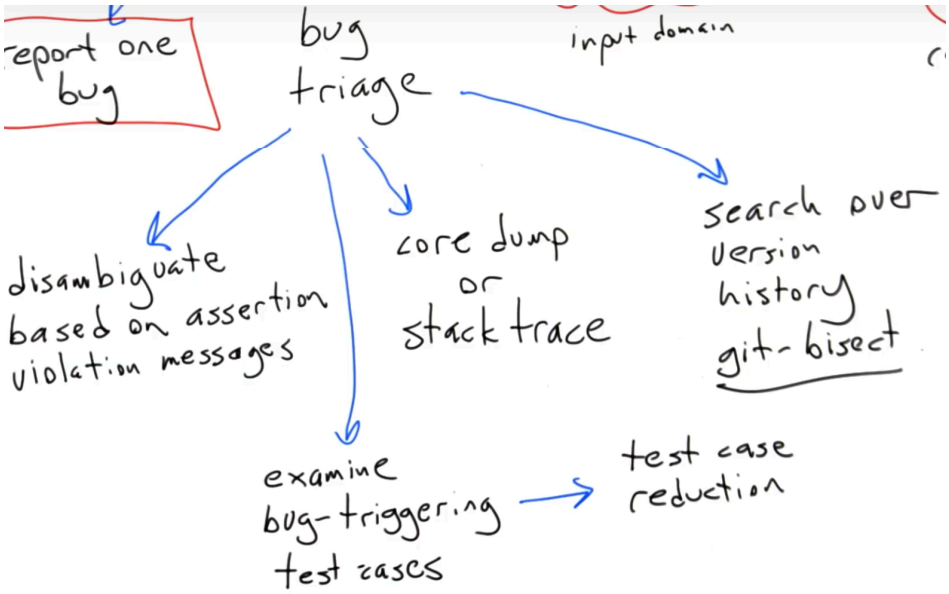
•For real software products we're forced to use a different strategy called **bug triage**.

3. Bug triage

- A bug triage is the process by which the severity of different bugs is determined, and we start to disambiguate between different bugs in order to basically try to get a handle on **which bugs we can report in parallel**.
- One of the **golden rules of reporting bugs** is that we don't report duplicate bugs.



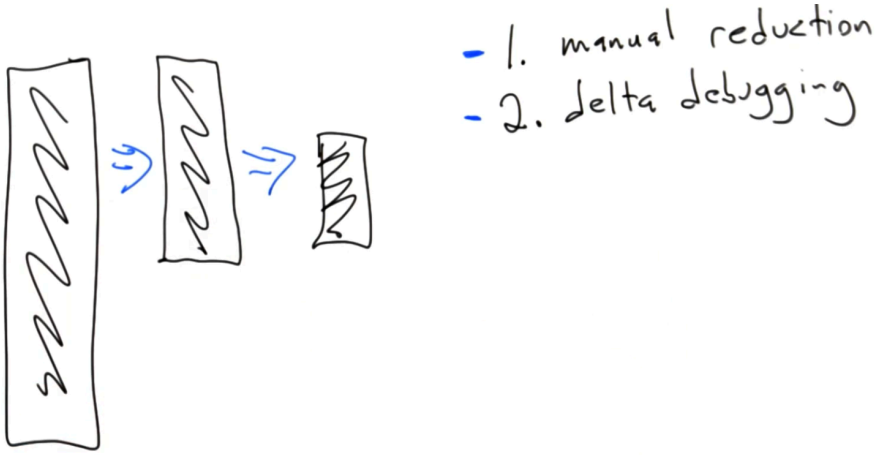
4. Difficulties with bug triage



- **Test-case reduction** or test-case minimization is an automated process of taking some large input that triggers a failure and turning it into a small input.

5. Test case reduction

- Usually many bugs can be triggered by small inputs, but often the inputs are huge.
- We can figure out, by hand, what part of the input causes the test case. We eliminate part of the input. Sometimes we do this in a smart way. For example, we might just know that some part of it is unlikely to trigger a crash, or we might just chop some of it out blindly and see if the smaller input triggers the test case. If it doesn't, then we go back to our original test case and try again. If it does, we proceed. If we're really lucky, at the very end of this process, we'll end up with a really small test case.



- **Delta debugging** is a tehchnique that automates this process. The framework takes our script and the test

input and automates this process in a loop. The loop terminates when the delta debugger can't reduce the input anymore, using built-in heuristics.

6. Delta debugging

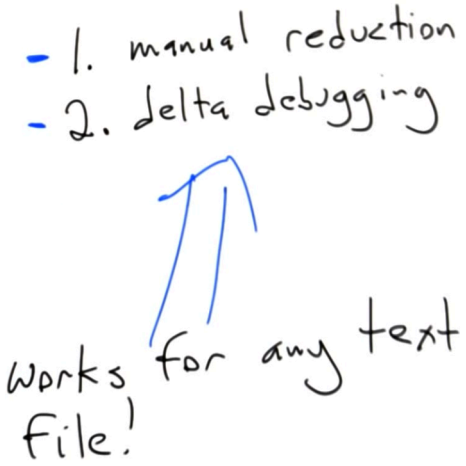
- In this section is presented a test case which makes GCC crash.

```
[regehr@dyson r48]$ current-gcc -O3 small.c
small.c:695:1: warning: large integer implicitly truncated to unsigned type [-Woverflow]
 static unsigned g_224 = 18446744073709551615UL;
 ^
small.c: In function 'func_52':
small.c:1342:8: warning: large integer implicitly truncated to unsigned type [-Woverflow]
        unsigned l_381 = 18446744073709551609UL;
        ^
small.c: In function 'func_59':
small.c:1537:8: warning: overflow in implicit constant conversion [-Woverflow]
        g_254 = 0x59846D02L;
        ^
small.c: In function 'func_65':
small.c:1630:3: warning: large integer implicitly truncated to unsigned type [-Woverflow]
   unsigned l_75 = 18446744073709551615UL;
   ^
small.c: In function 'func_52.isra.12':
small.c:1241:1: internal compiler error: in get_initial_def_for_induction, at tree-vect-loop.c:3222
 func_52 (unsigned char p_53, unsigned p_54)
 ^
Please submit a full bug report,
with preprocessed source if appropriate.
See <http://gcc.gnu.org/bugs.html> for instructions.
[regehr@dyson r48]$
```

- We run a delta debugger called godelta on it, i.e. an automatic script that detects if a particular program has this bug. The script is called test1.sh that compiles a program small.c at the particular optimization level -03 and then searches the output for this particular string get_initial_def_for_induction.

```
[regehr@dyson r48]$ godelta8 ./test1.sh small.c
```

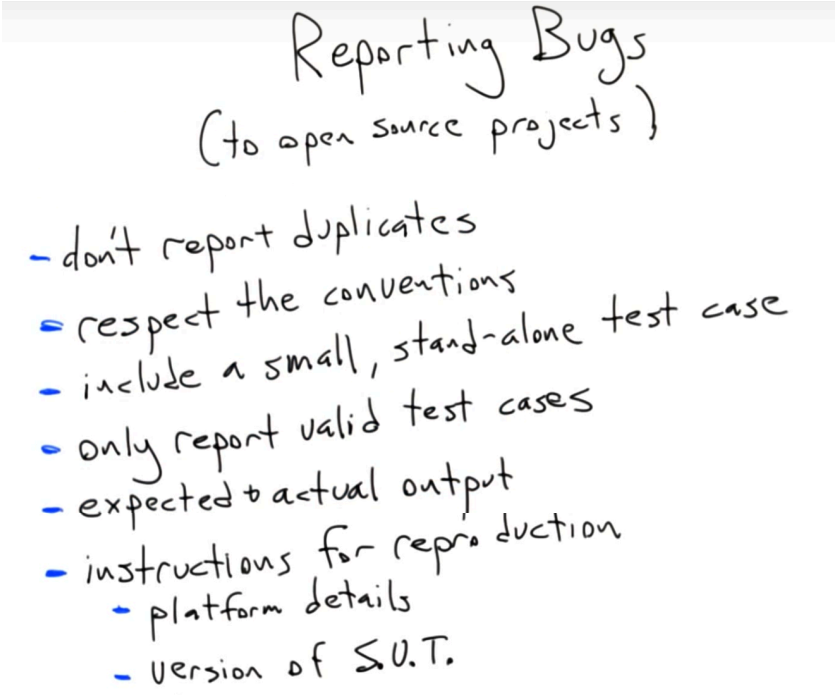
- The original program had nearly **40 KB**. Delta debugger got rid of junk and reduced to about **7 KB**.
- More about the analysed violated assertion and demo see the Udacity lesson 9.



- If we think back to the bounded queue data structure that we developed a random tester for, those particular random tests didn't have any external manifestation. They existed as data structures internal to Python. We can't do delta debugging because there's no text file, but what we could have done is before running the random tests through the queue, we could have saved them to a text file, i.e. just translating the operations that we were doing into strings, and then once saved on disk we can run the delta debugger and come up with a minimal set of queue API operations that cause the queue to do something wrong.
- **If we can do test-case reduction before doing the bug triage then the whole process becomes really a lot easier.**

7. Reporting bugs

- Let's assume that we created a random tester that starts crashing software. Instead of just being happy about what a successful effort we've had, we'll report the bugs to the maintainers so they can start improving the robustness of the software.



8. Example bug report

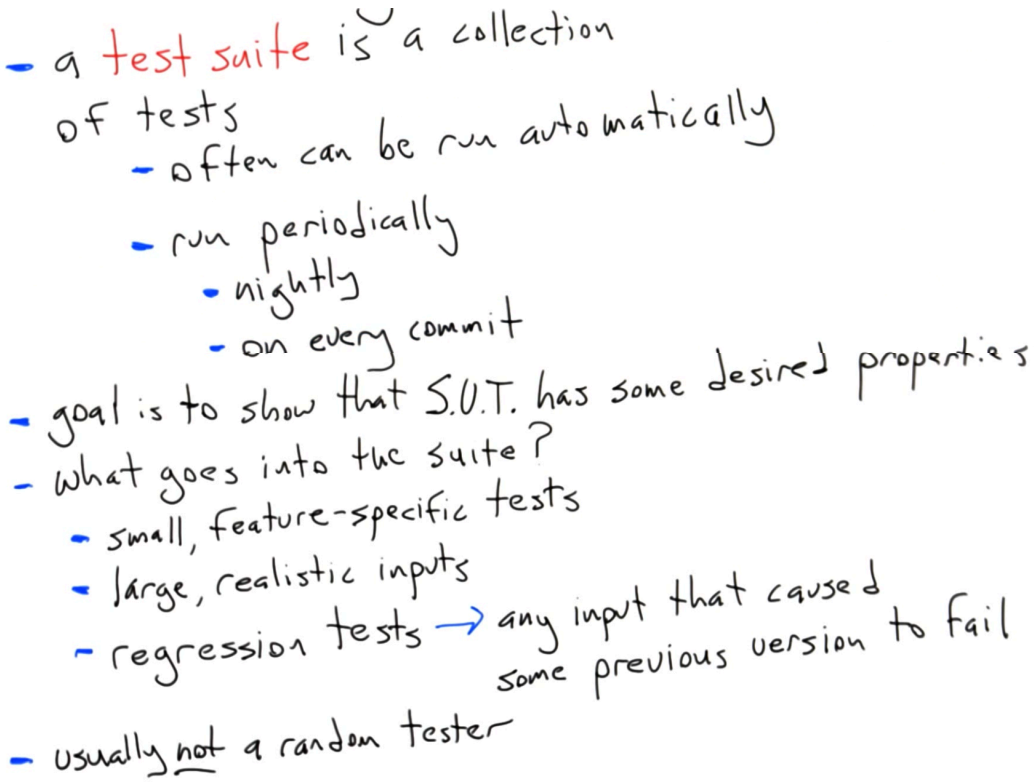
- In this section is presented and reported an LLVM bug.
- Instead of invoking the delta debugger for test case reduction is invoked a different tool, produced by the research group at School of Computing, University of Utah, called **C-Reduce** (https://github.com/csmith-project/creduce).

- C-Reduce is an extremely special-purposed delta debugger, which operates on exactly the same delta debugging ideas that the other tool operates by, and it has extra knowledge embedded in it about how to reduce C programs.

```
[regehr@dyson r47]$ creduce ./test1.sh small.c
```

- After 11 minutes of automated search, the output of the reduced test case for the same program used in section 6 (small.c) has **274 B**.
- More about the analysed violated assertion and demo see the Udacity lesson 9.

9. Building a test suite



- There are several reasons regression tests exist. The main one of which, is we want to make sure that the SUT doesn't regress, i.e. that it doesn't go back into a state in which it fails on a bug that we already fixed.
- Random tests are often non-deterministic, they don't have a clear correctness criterion and more importantly, random tests always have a possibility of showing us something new, i.e., they have the possibility of introducing a test case that we haven't seen before. Test suite is supposed to be predictable, to consist of things that we know to test. If all of a sudden, it starts containing new and different tests, then that's not necessarily good. So, for whatever combination of these reasons, random testing is often a separate activity.

10. Hard testing problems

- What can make testing really hard?
  - Lack of a specification
  - No comparable implementations
  - Big SUT
  - Large, highly structured input space
  - Non-determinism
  - Lots of hidden state
  - Lack of strong oracles
- Examples:
  - Large molecular simulation
  - Autopilot
  - JVM running on 32 cores for a month, using 40 GB of heap
- What to do:
  - Weak oracle
  - Try to make do with small, hand-checked test inputs
  - Rely on code inspections, formal methods, and other non-testing methods

11. Summary of testing principles

- Testers must want software to fail
- Testers are detectives: they must be observant for suspicious behaviour and anomalies in the SUT
- All available test oracles should be used in testing
- Test cases should contain values selected from the entire input domain
- Interfaces that cross a trust boundary need to be tested with representable values, not just those from the ostensible input domain
- A little brute force goes a long ways:
  - Sometimes, selected interfaces can be exhaustively tested
  - Almost everything else can be randomly tested
- Quality cannot be tested into bad software; in contrast, testable software has:
  - No hidden coupling, side channels
  - Few variables exposed to concurrent access
  - Few globals shared between modules
  - No pointer soup
- Code should be self checking, whenever possible, using lots of assertions; however:
  - These assertions are not used for error-checking

    - Assertions must never be side effecting
    - Assertions should never be trivial, or silly
- When appropriate, all 3 kinds of input should be used as a basis for testing:
    - APIs that are provided by the SUT, can be tested directly
    - APIs used by the SUT can be tested using fault injection
    - Non-functional inputs
- Failed coverage items do not provide a mandate to cover the failed items, but rather give clues to ways in which the tests are inadequate.

Notă[1]

[1]