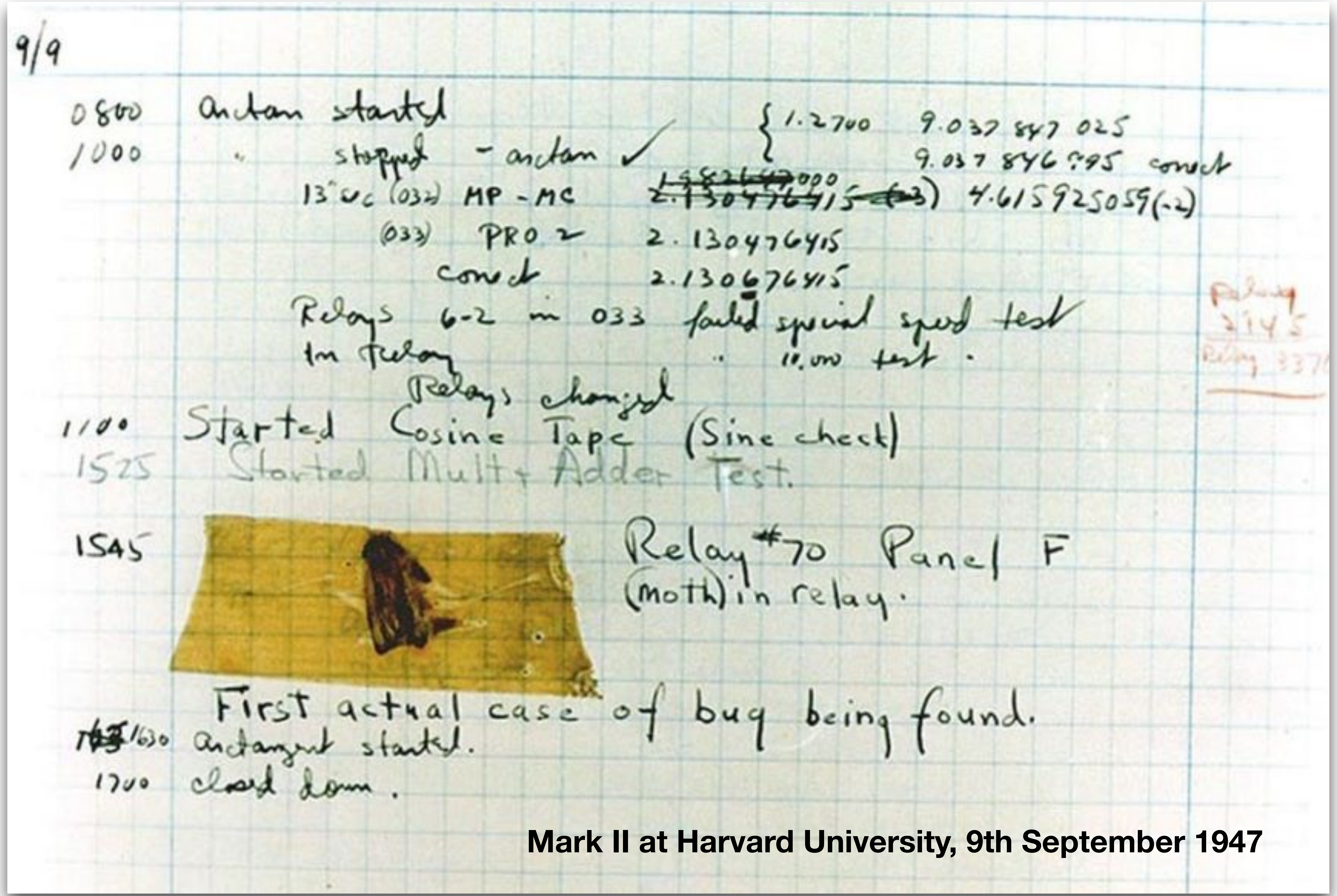
Debugging

CS350 Introduction to Software Engineering

Shin Yoo



9/9

0800 Arctan started

1000 " stopped - arctan ✓ $\left\{ \begin{array}{l} 1.2700 \quad 9.037\,847\,025 \\ \quad\quad\quad\quad\ 9.037\,846\,995 \text{ correct} \end{array} \right.$

13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (~~03~~) 4.615925059(-2)

(033) PRO 2 2.130476415

correct 2.130676415

Relays 6-2 in 033 failed special speed test

In relay " 11.000 test.

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F (moth) in relay.

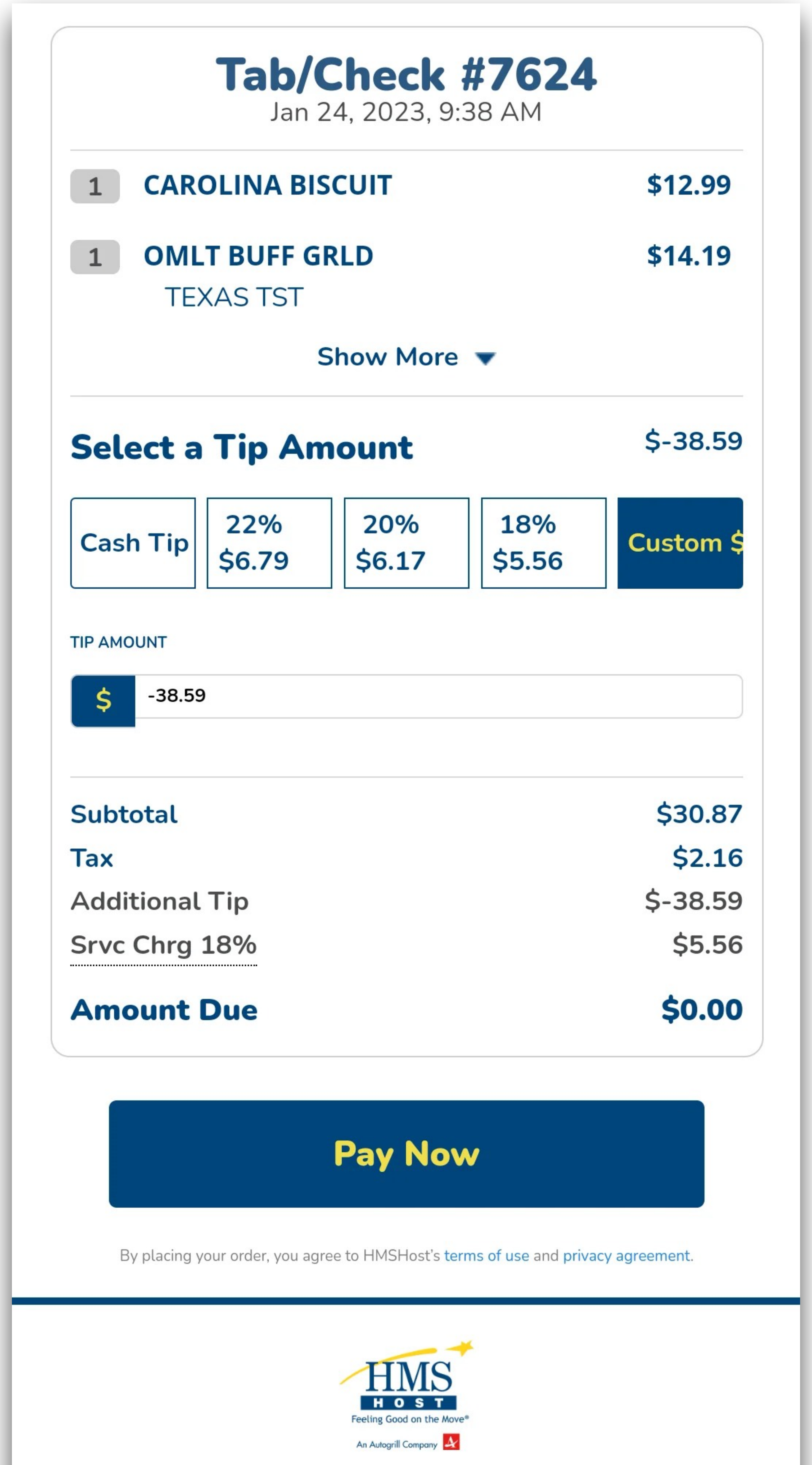
First actual case of bug being found.

~~1600~~ 1630 Antangent started.

1700 closed down.

Relay 2145
Relay 3370

Mark II at Harvard University, 9th September 1947



Tab/Check #7624
Jan 24, 2023, 9:38 AM

1	CAROLINA BISCUIT	$12.99
1	OMLT BUFF GRLD TEXAS TST	$14.19

Show More

Select a Tip Amount $-38.59

TIP AMOUNT
$ -38.59

Subtotal	$30.87
Tax	$2.16
Additional Tip	$-38.59
Srvc Chrg 18%	$5.56
Amount Due	**$0.00**

By placing your order, you agree to HMSHost's terms of use and privacy agreement.

https://www.reddit.com/r/interestingasfuck/comments/10ys42d/airport_software_allowed_negative_tip_to_cancel/

F-16 Problems (from Usenet net.aviation)

Bill Janssen <janssen@mcc.com>

Wed, 27 Aug 86 14:31:45 CDT

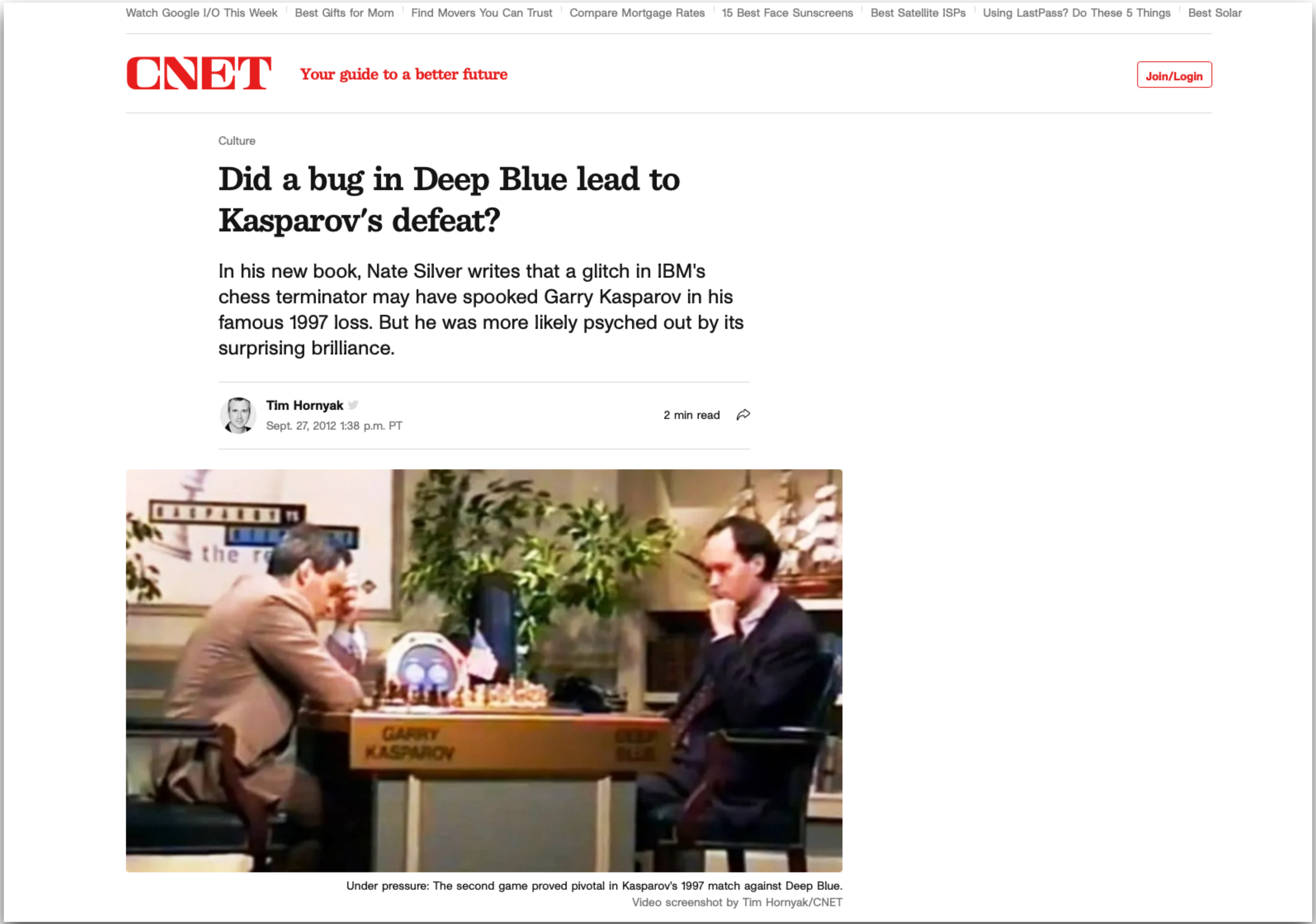
```
A friend of mine who works for General Dynamics here in Ft. Worth wrote some
of the code for the F-16, and he is always telling me about some
neato-whiz-bang bug/feature they keep finding in the F-16:

o Since the F-16 is a fly-by-wire aircraft, the computer keeps the pilot fro
  doing dumb things to himself. So if the pilot jerks hard over on the
  joystick, the computer will instruct the flight surfaces to make a nice an
  easy 4 or 5 G flip. But the plane can withstand a much higher flip than th
  So when they were 'flying' the F-16 in simulation over the equator, the
  computer got confused and instantly flipped the plane over, killing the
  pilot [in simulation].  And since it can fly forever upside down, it would
  do so until it ran out of fuel.

(The remaining bugs were actually found while flying, rather than in
simulation):
```

o One of the first things the Air Force test pilots tried on an early F-16 was to tell the computer to raise the landing gear while standing still on the runway. Guess what happened? Scratch one F-16. (my friend says there is a new subroutine in the code called 'wait_on_wheels' now...) [weight?]

o The computer system onboard has a weapons management system that will attempt to keep the plane flying level by dispersing weapons and empty fuel tanks in a balanced fashion. So if you ask to drop a bomb, the computer will figure out whether to drop a port or starboard bomb in order to keep the load even. One of the early problems with that was the fact that you could flip the plane over and the computer would gladly let you drop a bomb or fuel tank. It would drop, dent the wing, and then roll off.



Watch Google I/O This Week | Best Gifts for Mom | Find Movers You Can Trust | Compare Mortgage Rates | 15 Best Face Sunscreens | Best Satellite ISPs | Using LastPass? Do These 5 Things | Best Solar

CNET Your guide to a better future

Join/Login

Culture

Did a bug in Deep Blue lead to Kasparov's defeat?

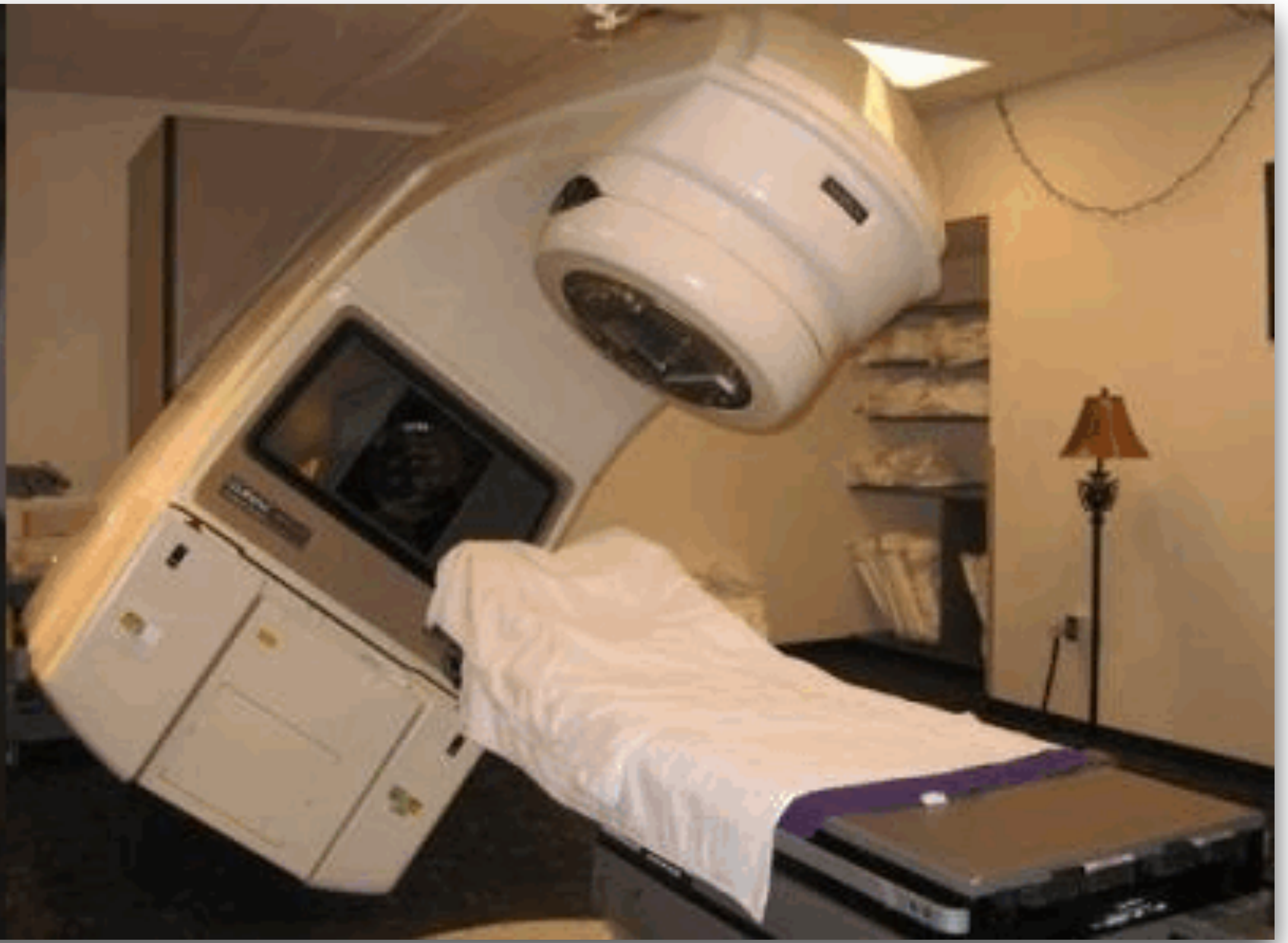
In his new book, Nate Silver writes that a glitch in IBM's chess terminator may have spooked Garry Kasparov in his famous 1997 loss. But he was more likely psyched out by its surprising brilliance.

Tim Hornyak
Sept. 27, 2012 1:38 p.m. PT

2 min read

Under pressure: The second game proved pivotal in Kasparov's 1997 match against Deep Blue.
Video screenshot by Tim Hornyak/CNET

https://www.cnet.com/culture/did-a-bug-in-deep-blue-lead-to-kasparovs-defeat/



https://en.wikipedia.org/wiki/Therac-25

A couple of more classic bug stories

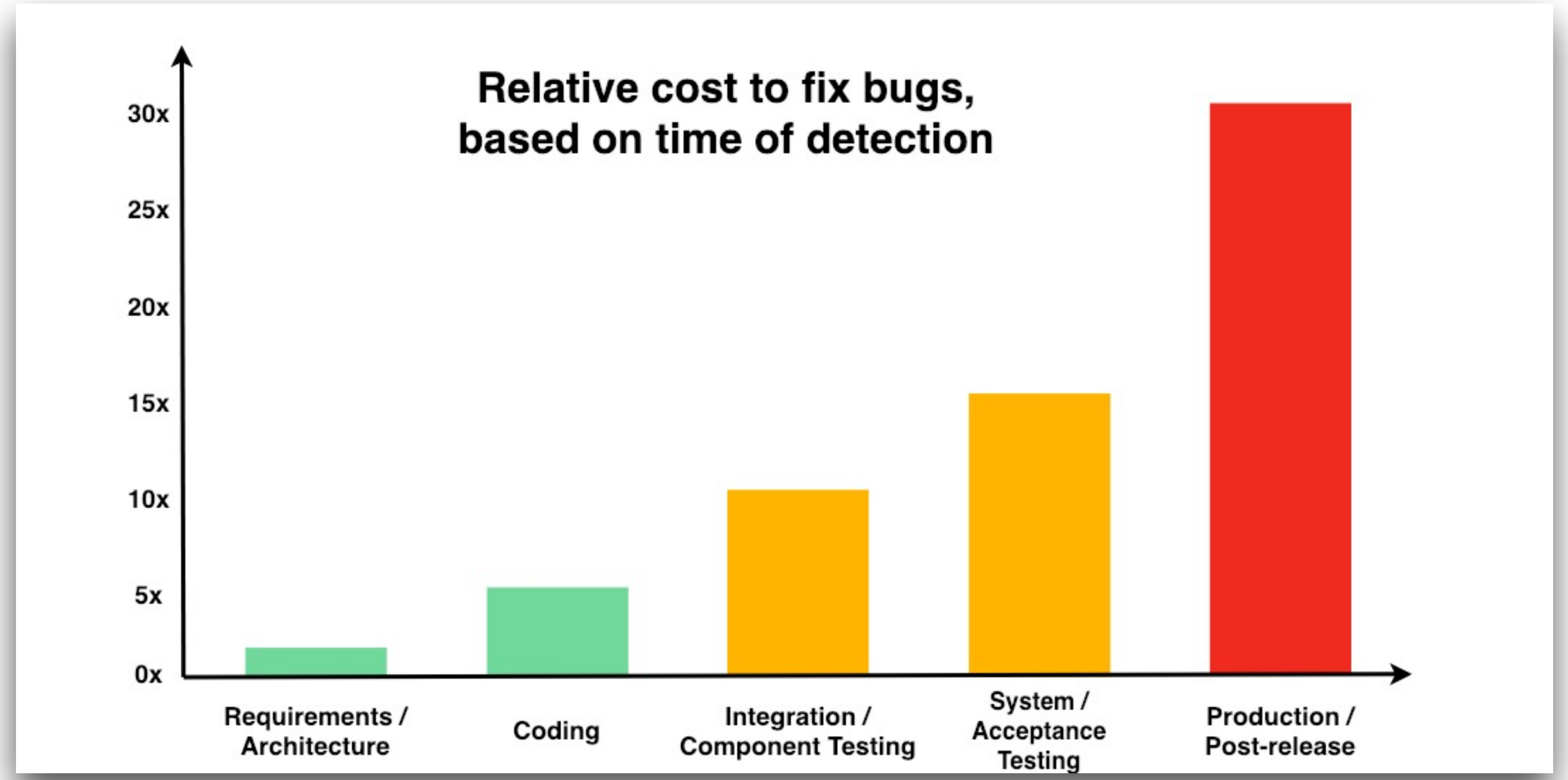
- Emails being sent to only up to 500 miles: https://www.ibiblio.org/harris/500milemail.html
- A car that is allergic to vanilla ice cream: https://www.digitalrepublik.com/digital-marketing-newsletter/2015/05/10/my-car-does-not-start-when-i-buy-vanilla-ice-cream-said-a-man-to-general-motors/

Bugs

- Developer makes a mistake and creates a defect/fault in the source code.
- During execution, this leads to an infected program state.
- The infected state is propagated to observable behaviour and becomes a failure.

Debugging

- The process of finding and resolving bugs (wikipedia)
- Goes hand in hand with testing; the earlier, the better.



NIST Report: The Economic Impacts of Inadequate Infrastructure for Software Testing, 2002
https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf

Seven Steps in Debugging

taken from “Why programs fail: a guide to systematic debugging” by A. Zeller

- **T**rack the problem in the database: record, deduplicate, manage
- **R**eproduce the failure
- **A**utomate and simplify the test
- **F**ind possible infection origins
- **F**ocus on the most likely origin (known infections, causes in state, code, and input, anomalies, and code smells)
- **I**solate the infection chain
- **C**orrect the defect

Failure Reproduction

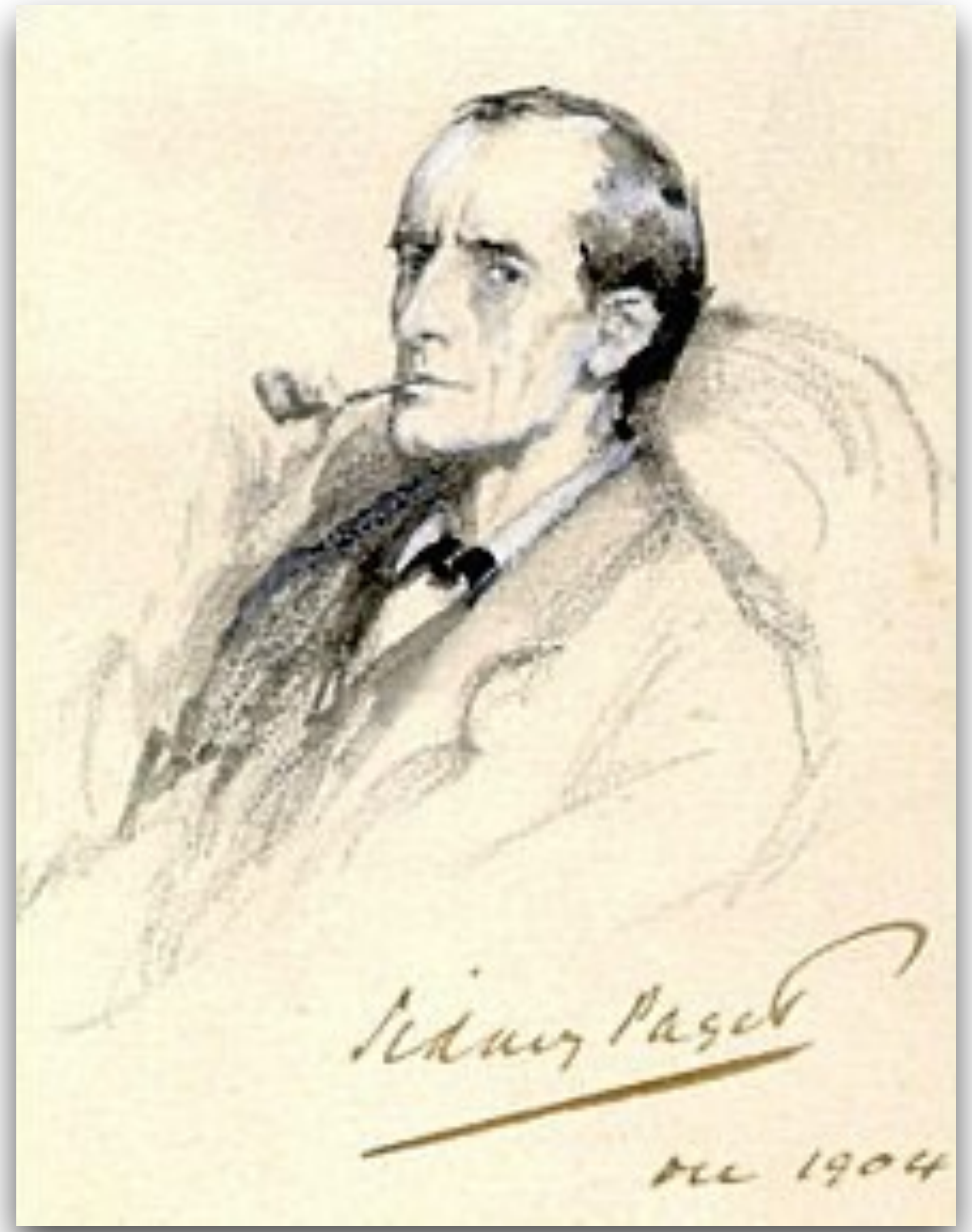
- If the bug is detected by one of your test cases, chances are you can already reproduce it. If you cannot reproduce the behaviour of your own tests, it may be flaky, or there may be an unknown factor in your testing.
- If the bug is reported by someone else:
  - Ideally, the bug report should contain steps required for reproduction, even if it is only in natural language. You can reconstruct this as a test case.
  - If there is no such information, you still have to reproduce the reported behaviour.

Automated Failure Reproduction

- Note that failure reproduction also requires a test oracle, i.e., we need a mechanism that can tell us whether we have reproduced the same bug or not.
    - Implicit oracle: we have techniques that can reproduce crashes automatically - intuitively, we try to automatically generate the same input that results in a similar call-stack or execution trace as the crash.
    - Explicit oracle: still difficult - people are investigating whether ML can help with the oracle problem, but it is still very early days. There are also research that aim to reduce human efforts in oracle evaluation.
- “Cannot reproduce” is one of the common answers to bug reports…

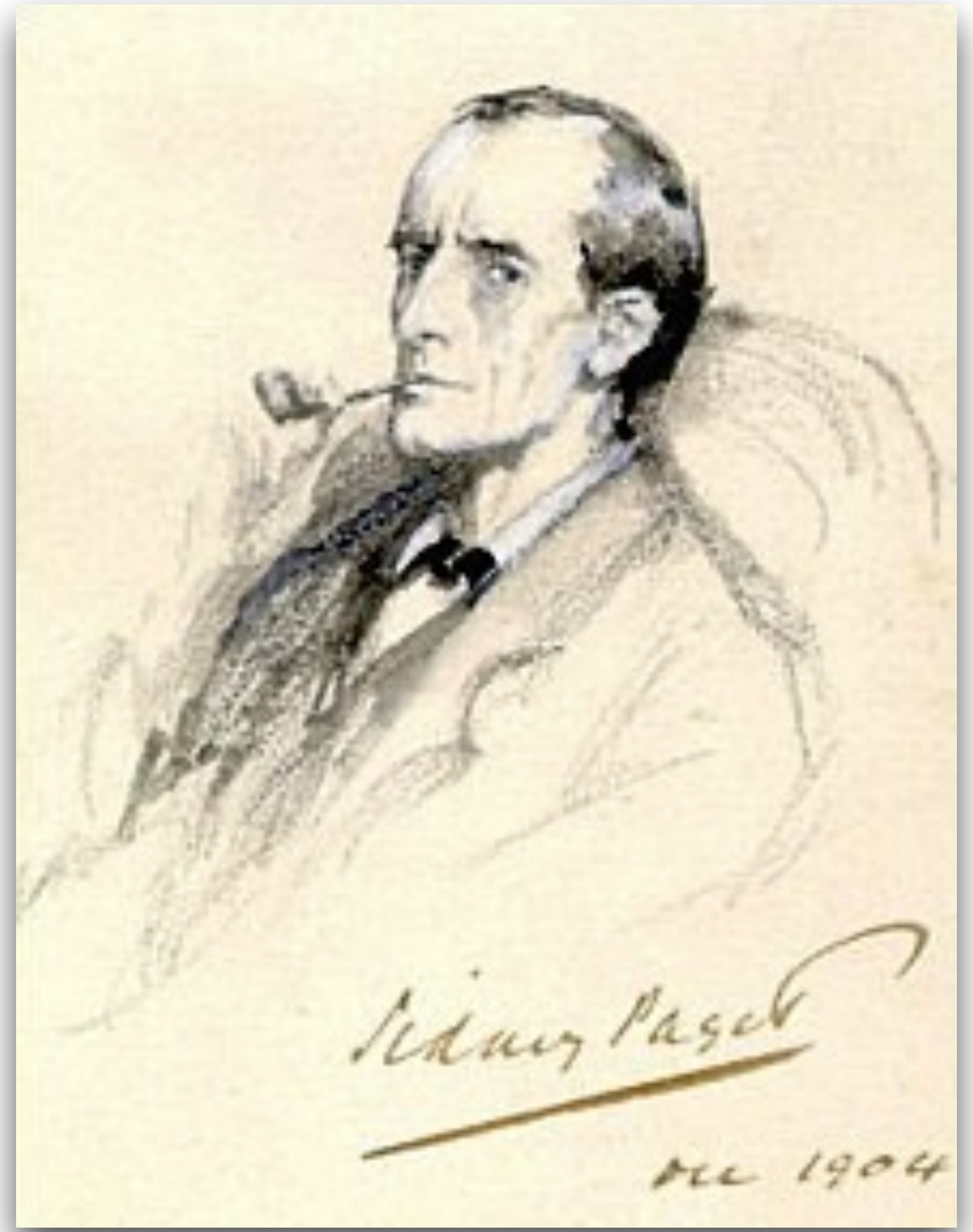
Finding the Origins of Infection

Don’t do the ritualistic debugging



Finding the Origins of Infection

- "Once you eliminate the impossible, whatever remains, no matter how improbable, must be the truth."
- Essentially debugging is the art of elimination. To be successful, you need a couple of tricks:
    - Thorough observation: you should be able to observe any aspect of the system that you suspect
    - Systematic elimination: analyse everything in an organised approach



Scientific Debugging - A. Zeller

A framework for systematic “narrowing down”

- Adopt the process of scientific discovery
  - Observe the failure
  - Hypothesize a potential cause that matches the observation
  - Use the hypothesis to make a prediction
  - Test the hypothesis by performing an experimentation
  - If the hypothesis is backed by the experimentation, debug is successful; otherwise, repeat with an alternative hypothesis

Lower Level Debugging Techniques and Aids

Observation: Tracing

- AKA “printf debugging”: easy, intuitive, effective… we have all done this.
- However, printf can also be problematic:
  - In some environment, the I/O itself is very complicated or even unavailable.
  - We are modifying program behaviour to observe something: this can result in so-called Heisenbugs.
  - If you want to modify what you want to observe, you need to insert new statements, or modify and build the program, which is costly.

Observation: Debuggers

- Debuggers allow us to execute the target program under controlled condition, and to pause the execution and make observations.
- Consider, the example of python debugger, pdb.
- We will immediately pause the execution at the beginning. Then we will conditionally pause next time when we are about to attempt a division by zero.

```python
import pdb
breakpoint()

vals = [2, 4, 7, 4, 3, 0, 9]

def inverse(x):
    return 1 / x

for val in vals:
    print(f"Inverse of {val} is {inverse(val)}.")
```

Heisenbugs

Observation itself can affect the bug (examples from Zeller)

```
int f(){
    int i;
    return i
}
```

Okay in real executions, weird in debuggers… why?

A problem goes away if you observe it using printf, but comes back if you do not observe… why?

```
printf("$d\n", suspicious_variable);
```

Tracing vs. Debugger

- Both have pros and cons.
- Modern IDEs come equipped with very powerful debuggers: get familiar with the basics of the debugger of your favourite language.
- Printf is simple and intuitive, but use proper logging whenever possible.

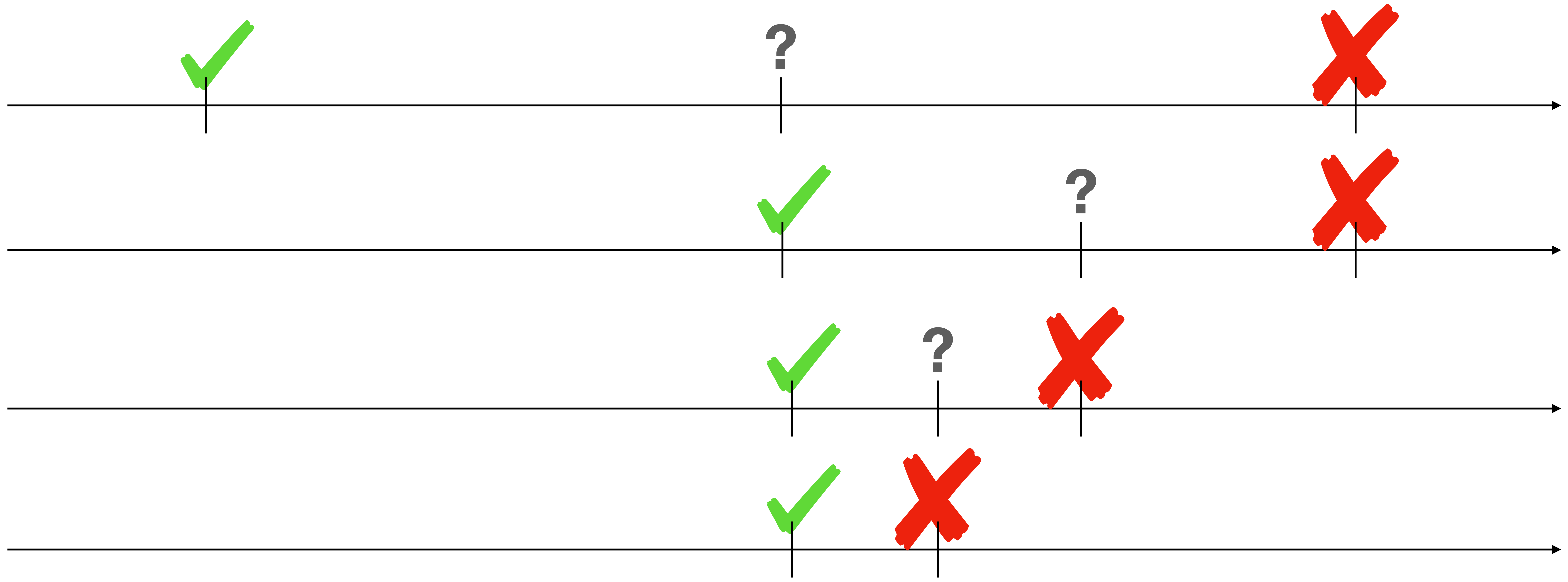
How to simplify and/or narrow down origins?

- Along the temporal axis
  - When did this bug come into the codebase? (bisection)
- Along the spatial axis
  - Where in the input is the trigger? (input minimisation)
  - Where in the source code is the actual bug?(fault localisation)

Bisection

(Remember VCS: git bisect?)

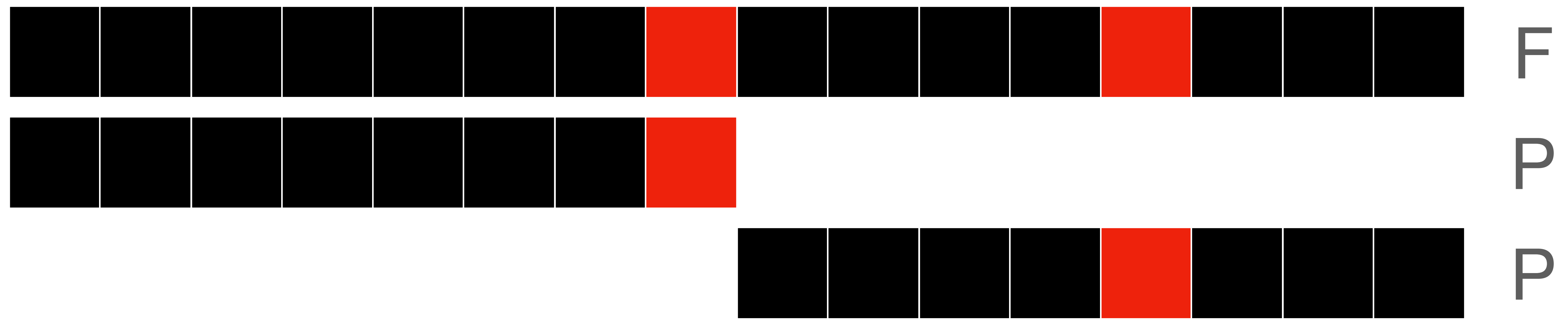
- Perform binary search between the last known **good** point, and the latest **bad** point



Delta Debugging

A systematic input minimisation technique by Zeller

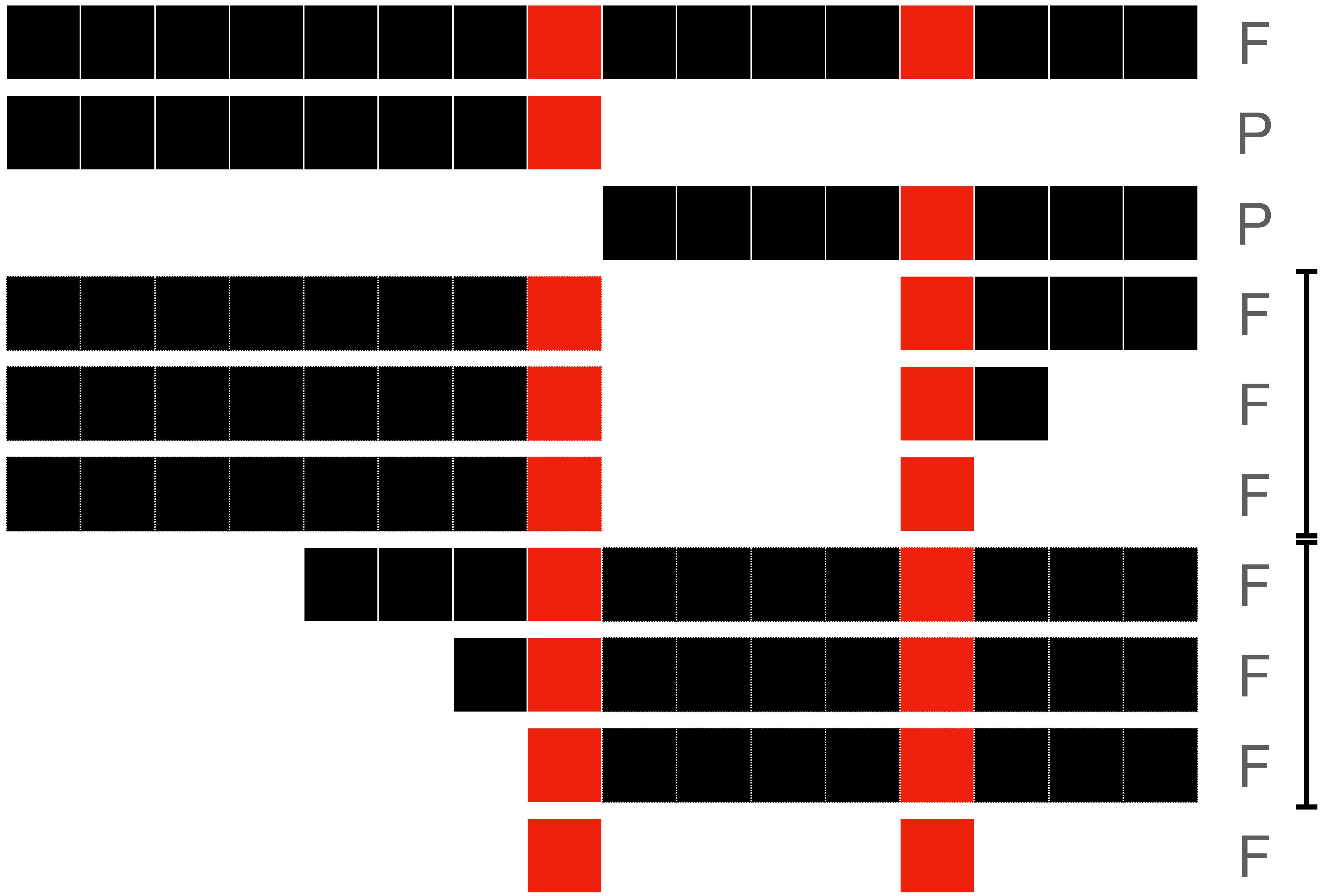
- Bisection is narrowing down a segment of time
- Delta Debugging is narrowing down a segment of input that causes the problem. However, unlike time segments, problematic inputs can be discontinuous.



Delta Debugging

- Line 1: if there is a single input left, return it.
- Line 2-7: see if program still fails with halves of the given input - if it does, continue halving recursively
- Line 8-10: otherwise, make two recursive calls
    - First: keep the first half of the given input, and apply DD to the second half
    - Second: keep the second half of the given input, and apply DD to the first half

```
$\mathrm{DD}(P, \{i_1, \ldots, i_n\})$
(1)     $\textbf{if } n == 1 \textbf{ then return } i_1$
(2)     $P_1 = \left(P + \left\{i_1, \ldots, i_{\frac{n}{2}}\right\}\right)$
(3)     $P_2 = \left(P + \left\{i\frac{n}{2} + 1, \ldots, i_n\right\}\right)$
(4)     $\textbf{if } P_1 \text{ fails}$
(5)         $\textbf{return } \mathrm{DD}\left(P, \left\{i_1, \ldots, i_{\frac{n}{2}}\right\}\right)$
(6)     $\textbf{else if } P_2 \text{ fails}$
(7)         $\textbf{return } \mathrm{DD}\left(P, \left\{i\frac{n}{2} + 1, \ldots, i_n\right\}\right)$
(8)     $\textbf{else}$
(9)         $\textbf{return } \mathrm{DD}\left(P_2, \left\{i_1, \ldots, i_{\frac{n}{2}}\right\}\right) +$
(10)        $\mathrm{DD}\left(P_1, \left\{i_{\frac{n}{2}+1}, \ldots, i_n\right\}\right)$
```


F
P
P
F
F
F
F
F
F
F

Fault Localisation

- Static & dynamic analysis that aims to locate where the fault is.
- Typically statistical approach (i.e., they are heuristics):
    - Spectrum Based Fault Localisation: if a statement is executed more frequently by failing tests, and less frequently by passing tests, it is more suspicious.
    - Information Retrieval Based Fault Localisation: if a file or a method is lexically more similar to the bug report, it is more suspicious.
    - Mutation Based Fault Localisation: if mutating location X produces test results that are similar to the current failure, X is more suspicious.
- Naturally, many advanced machine learning approaches have been proposed.

Spectrum Based Fault Localisation

- Program Spectrum: for each structural unit (i.e. statements or branches), summarise the test result AND coverage into a tuple of the following four numbers
  - ep: # of test cases that execute this unit and pass
  - ef: # of test cases that execute this unit and fail
  - np: # of test cases that do not execute this unit and pass
  - nf: # of test cases that do not execute this unit and fail

Spectrum Based Fault Localisation

Structural Elements	Test t_1	Test t_2	Test t_3	Spectrum e_p	e_f	n_p	n_f	Tarantula	Rank
s_1	●			1	0	0	2	0.00	9
s_2	●			1	0	0	2	0.00	9
s_3	●			1	0	0	2	0.00	9
s_4	●			1	0	0	2	0.00	9
s_5	●			1	0	0	2	0.00	9
s_6	●		●	1	1	0	1	0.33	4
s_7 (faulty)		●	●	0	2	1	0	1.00	1
s_8	●	●		1	1	0	1	0.33	4
s_9	●	●	●	1	2	0	0	0.50	2
Result	P	F	F						

$$\text{Tarantula} = \frac{\frac{e_f}{e_f+n_f}}{\frac{e_p}{e_p+n_p}+\frac{e_f}{e_f+n_f}}$$

Repair

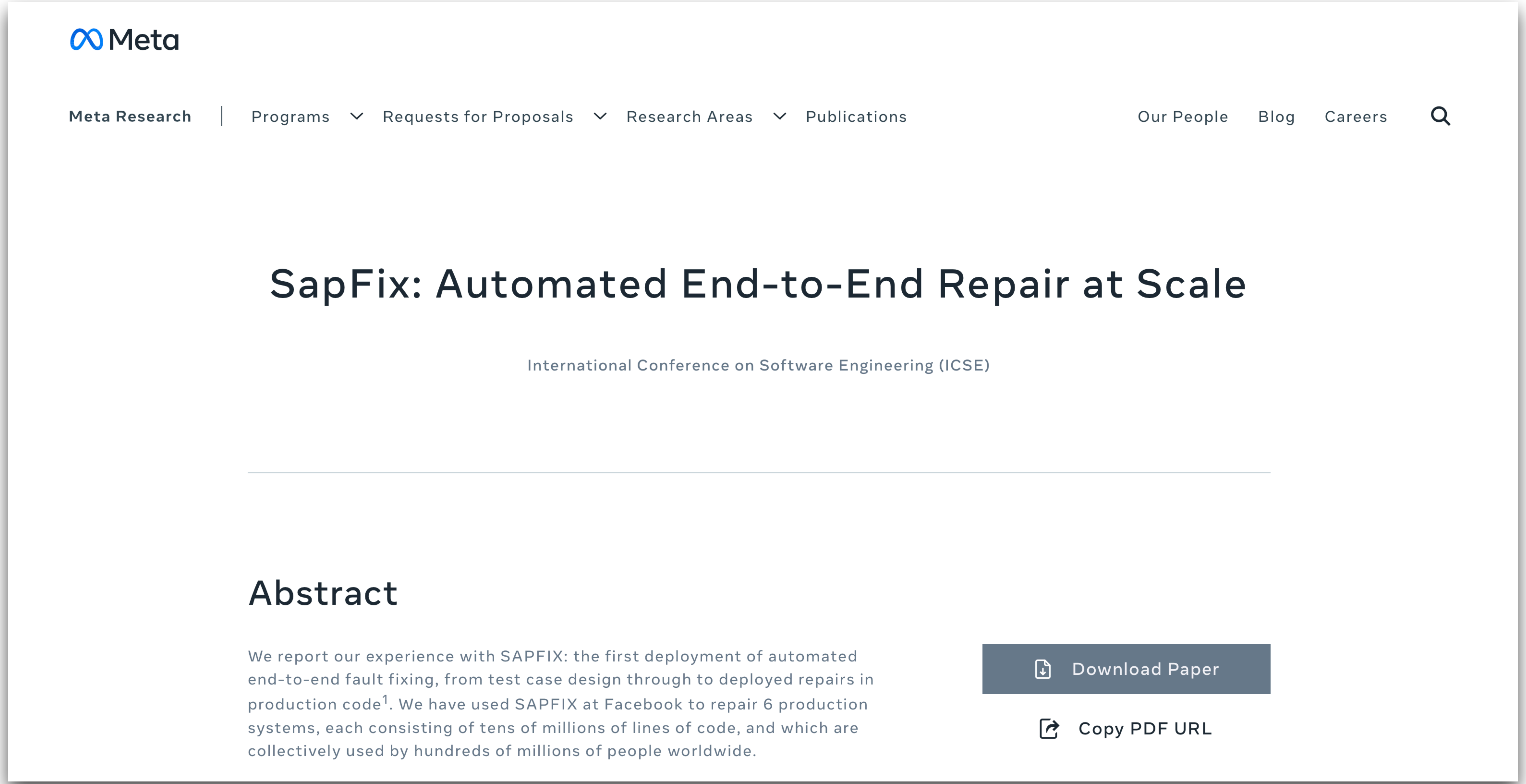
- Once you have a theory of the bug (via scientific debugging), you can also design a fix.
- Good practice of writing a patch (a fix):
    - Close the loop that started with “Track”: go back to the bug report if there is one, and close the issue. Explicitly link the “closure” with the bug fixing commit.
    - Patch should be maintainable: use appropriate comments and documentation.
    - A fix should be accompanied by a test, to avoid regression.
- Can we automate the patching itself?

Automated Program Repair

- In theory, fully automated, perfectly correct repair is not possible due to the oracle problem, as well as the undecidability of program semantics.
- However, there are still many fixes that we can find **automatically**!
- GenProg (2009)
    - Uses fault localisation techniques to identify likely targets to patch
    - Apply random edits (copying & inserting a statement from somewhere else, swapping two statements, deleting a statement) until test results gradually improve (okay, in reality there is an algorithm called genetic programming).
    - If no test fails, you have a candidate patch!

Automated Program Repair

- Some of the changes we apply to our source code are… *typical* or *repetitive*.
- "Plastic Surgery Hypothesis": changes to a codebase contain snippets that already exists in the codebase at the time of the change. (Barr et al., FSE 2014)
- Template Based Program Repair
    - Collect frequent code modifications as templates (e.g., adding a check for null pointer)
    - Find applicable templates for a given failure, apply, and validate using test cases



SapFix: Automated End-to-End Repair at Scale

International Conference on Software Engineering (ICSE)

Abstract

We report our experience with SAPFIX: the first deployment of automated end-to-end fault fixing, from test case design through to deployed repairs in production code[1]. We have used SAPFIX at Facebook to repair 6 production systems, each consisting of tens of millions of lines of code, and which are collectively used by hundreds of millions of people worldwide.

https://research.facebook.com/publications/sapfix-automated-end-to-end-repair-at-scale/

Summary

- Debugging should be a systematic process of elimination.
- Adopt scientific methods; there are various automated supporting techniques.
- Automated Program Repair is growing mature.