ECS 261
Lecture 3:

Intro to interactive verification

Plan

Introduction to interactive program verification and why it matters

(And the second half of the course)

(Slides today; back to live coding next time on Wednesday)

We know about

- Writing specifications (Hypothesis and Z3)
- Proving specifications correct (Z3)

(Really needs new logo)

We know about

- Writing specifications (Hypothesis and Z3)
- Proving specifications correct (Z3)

Main limitations of Z3?

https://forms.gle/uXELPFiRY85kb97Y6

Example

(from a recent unrelated project)

https://pastebin.com/D1cX6egj

(time to file an issue report)

Interactive verification

Basically a more powerful version of the above:

- We can write more general specifications
- We can write the proofs ourselves - don’t need to rely on the tools terminating or finding the proof automatically
- We can incorporate verified code into bigger projects

^^^ more work + more effort = more payoff

Why use formal verification?

So, you've written your code. You've tested it, and it seems to be working the way you expect.

> It's a lot of work to write specifications!
>
> It's a lot of work to prove specifications!

So when might you want to go the extra mile and do all this extra work?

Answer

Interactive verification is especially useful in cases where:

1. **Correctness is critical to your application**
2. **Security**
3. **A bug is very expensive or catastrophic**

1. Correctness is critical

If the software fails, some very serious consequence will occur

Pentium bug

Intel, 1994: Bug in floating point

$$\frac{4{,}195{,}835}{3{,}145{,}727} = 1.333739068902037589$$

Pentium bug

Intel, 1994: Bug in floating point

- December 1994: Intel **recalls** all Pentium processors
- $475 million in losses

Incident led to renewed interest in formal verification:

today, chip design at companies like Intel and IBM is

validated by formal methods prior to deployment

1. Correctness is critical

If the software fails, some very serious consequence will occur

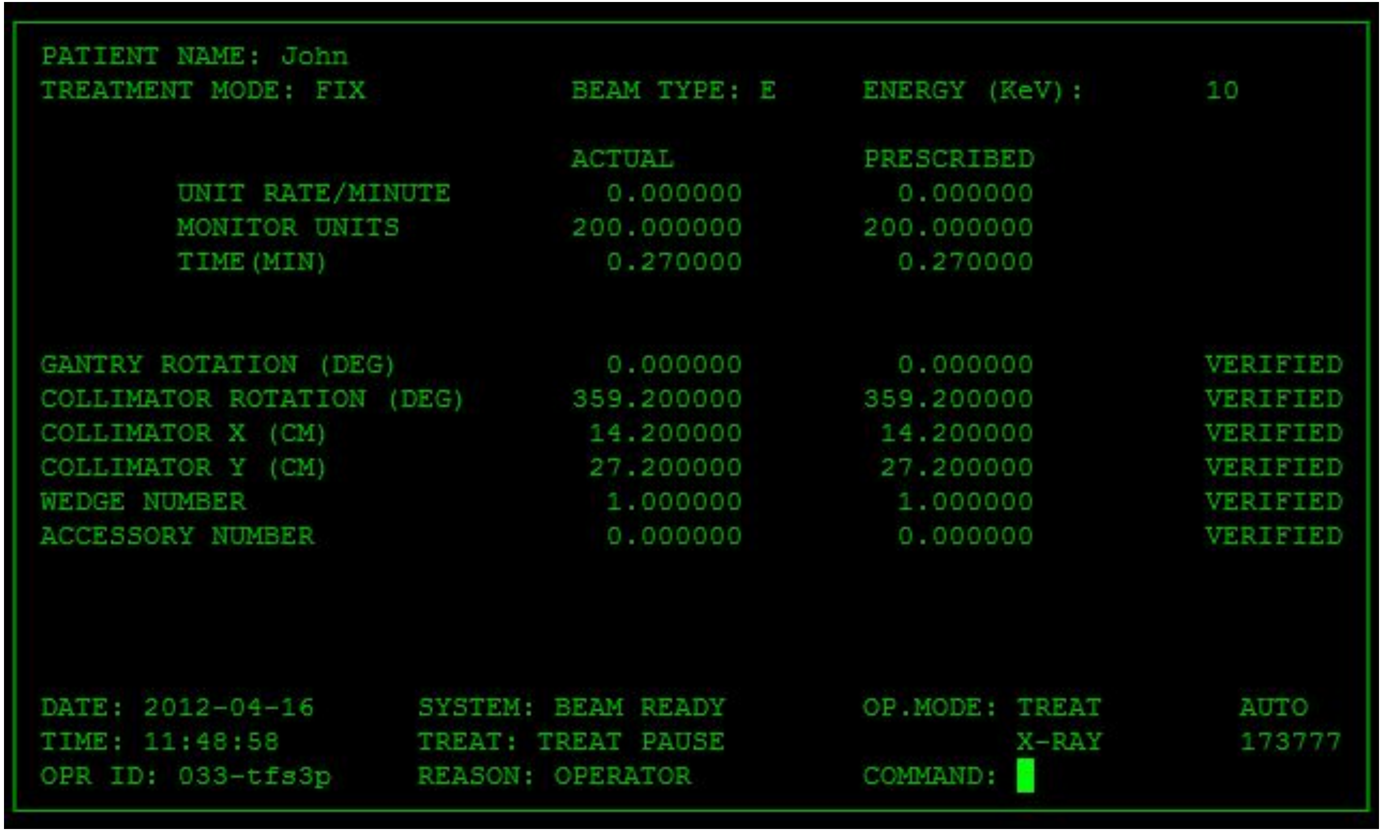
Therac-25

one of the most (in)famous software bugs in history

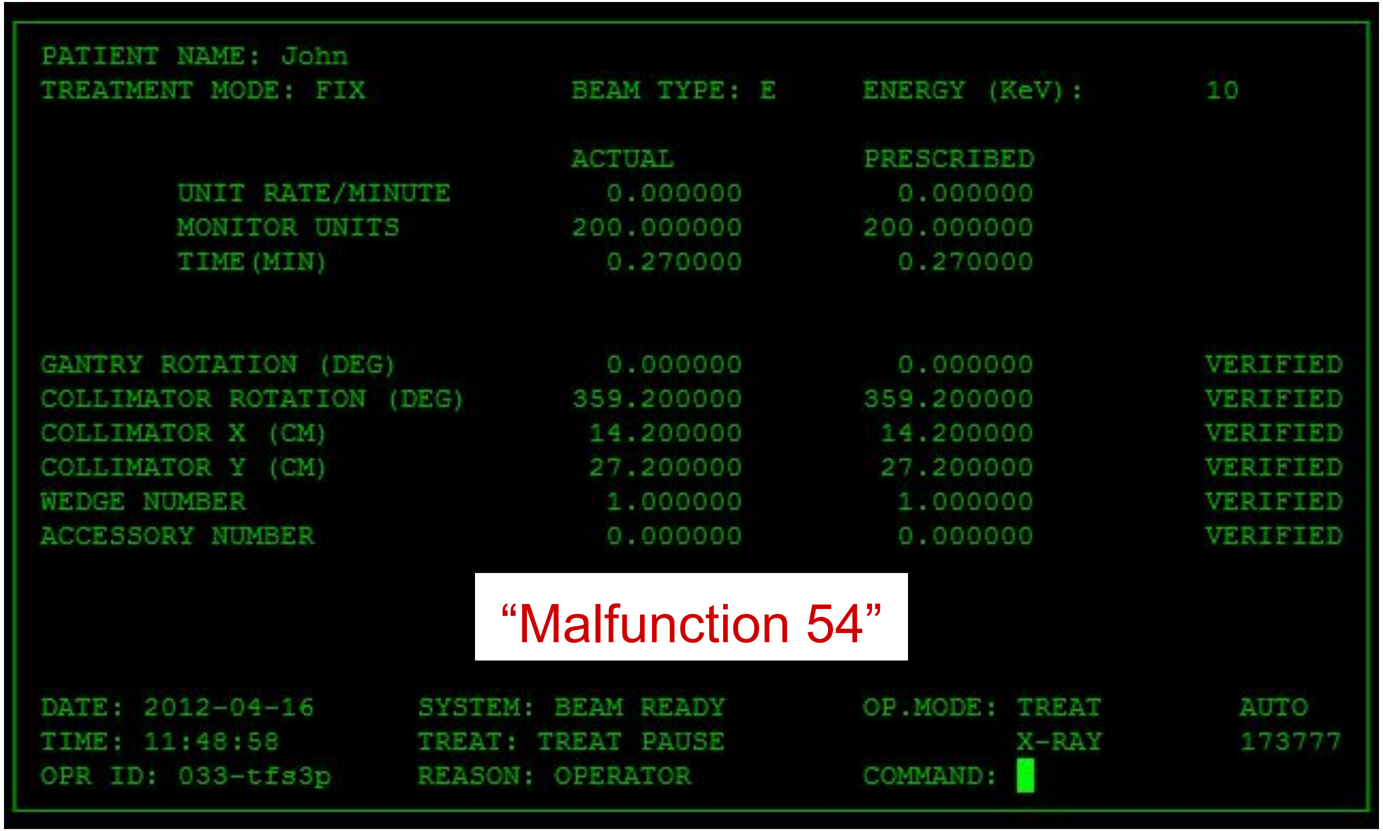
Radiation therapy machine (1985-1987)

- Under seemingly random conditions it would give 100+x the intended radiation dose to patients
- manufacturers repeatedly denied any fault and the machine's use continued even after the first overdoses
- At least 6 serious incidents, 3 deaths

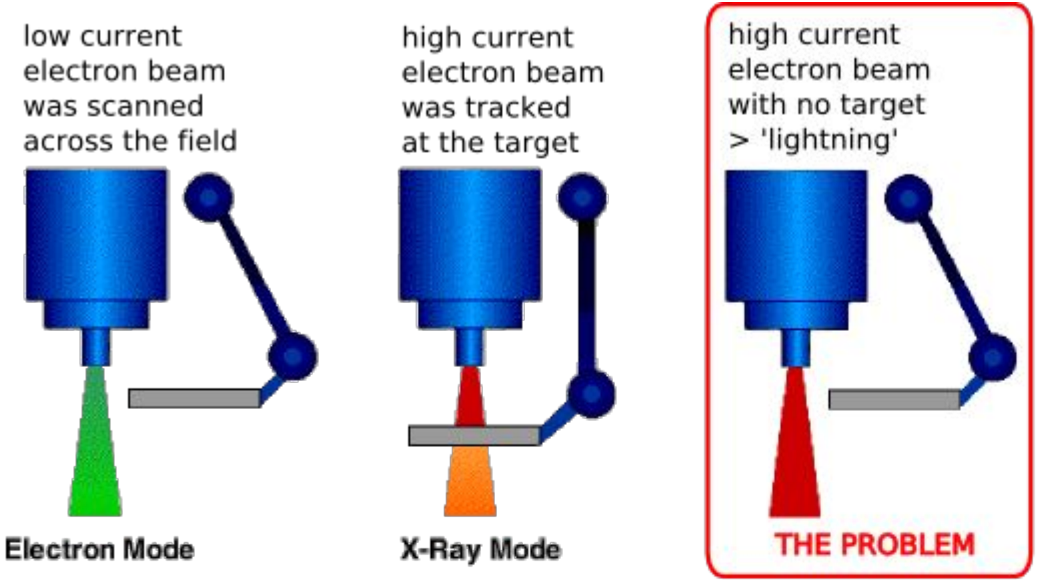
Therac-25


PATIENT NAME: John
TREATMENT MODE: FIX
BEAM TYPE: E
ENERGY (KeV): 10
ACTUAL
PRESCRIBED
UNIT RATE/MINUTE 0.000000 0.000000
MONITOR UNITS 200.000000 200.000000
TIME (MIN) 0.270000 0.270000
GANTRY ROTATION (DEG) 0.000000 0.000000 VERIFIED
COLLIMATOR ROTATION (DEG) 359.200000 359.200000 VERIFIED
COLLIMATOR X (CM) 14.200000 14.200000 VERIFIED
COLLIMATOR Y (CM) 27.200000 27.200000 VERIFIED
WEDGE NUMBER 1.000000 1.000000 VERIFIED
ACCESSORY NUMBER 0.000000 0.000000 VERIFIED
DATE: 2012-04-16
SYSTEM: BEAM READY
OP.MODE: TREAT
AUTO
TIME: 11:48:58
TREAT: TREAT PAUSE
X-RAY
173777
OPR ID: 033-tfs3p
REASON: OPERATOR
COMMAND:

Therac-25


PATIENT NAME: John
TREATMENT MODE: FIX
BEAM TYPE: E
ENERGY (KeV): 10
ACTUAL
PRESCRIBED
UNIT RATE/MINUTE 0.000000 0.000000
MONITOR UNITS 200.000000 200.000000
TIME (MIN) 0.270000 0.270000
GANTRY ROTATION (DEG) 0.000000 0.000000 VERIFIED
COLLIMATOR ROTATION (DEG) 359.200000 359.200000 VERIFIED
COLLIMATOR X (CM) 14.200000 14.200000 VERIFIED
COLLIMATOR Y (CM) 27.200000 27.200000 VERIFIED
WEDGE NUMBER 1.000000 1.000000 VERIFIED
ACCESSORY NUMBER 0.000000 0.000000 VERIFIED
“Malfunction 54”
DATE: 2012-04-16
TIME: 11:48:58
OPR ID: 033-tfs3p
SYSTEM: BEAM READY
TREAT: TREAT PAUSE
REASON: OPERATOR
OP.MODE: TREAT
X-RAY
COMMAND:
AUTO
173777

Therac-25



low current
electron beam
was scanned
across the field
high current
electron beam
was tracked
at the target
high current
electron beam
with no target
> 'lightning'
Electron Mode
X-Ray Mode
THE PROBLEM

Therac-25

The bug was detectable in software!

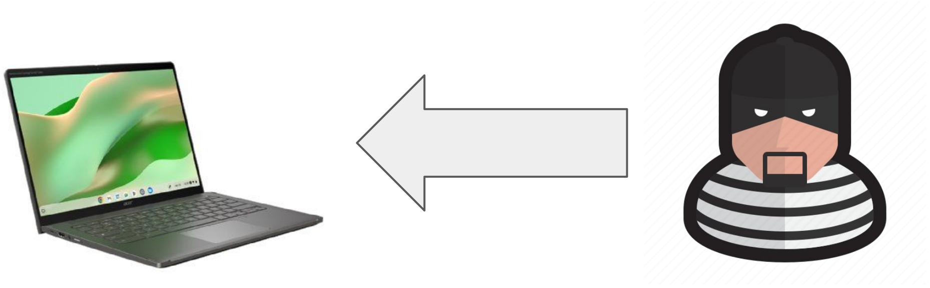
Malfunctions/errors were common when operating the terminal; operators learned to ignore them

Would verification help?

Yes: by making a known bad state unreachable

2. Security

If the software is vulnerable to attack, you may not have considered all the ways it could be exploited



Low-level cryptographic libraries

- if these are incorrect, it can take down the whole security foundation of the internet!
- Signal messaging app: verification effort for core messaging protocol going back to 2017

Low-level cryptographic libraries

AWS-LibCrypto:

- open source SSL/OpenSSL implementation that is proved using Coq, HOLLight, and other tools.
- Report

Other misc examples

Galois, inc. has several projects in this area including the

SAW verification tools and the

Cryptol domain-specific language

Access control bugs

Expose critical customer or user data to malicious actors!

Access control bugs

Cloud providers

- One serious bug would be enough to destroy trust in a provider

AWS is investing millions in verification tools (including using Z3 and Dafny) for AWS S3 and IAM, AWS Encryption SDK, and other projects)

3. Cost

A bug is very expensive or catastrophic for your company/organization

Other examples: blockchain technology

https://immunefi.com/immunefi-top-10/

Top vulnerabilities in smart contracts

“The Beanstalk Logic Error Bugfix Review showcases an example of a missing input validation vulnerability. The Beanstalk Token Facet contract had a vulnerability in the transferTokenFrom() function, where the msg.sender’s allowance was not properly validated during an EXTERNAL mode transfer. This flaw allowed an attacker to transfer funds from a victim’s account who had previously granted approval to the Beanstalk contract.”

Lots of startups, e.g.

- Cubist

  https://cubist.dev/about

- Veridise

  https://veridise.com/

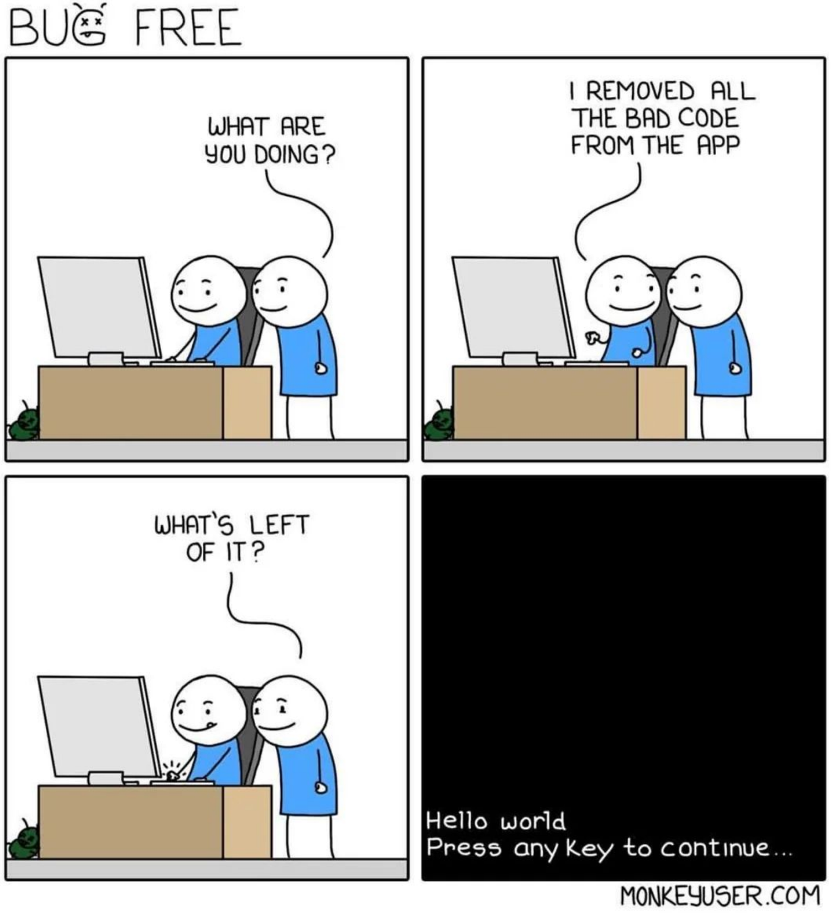
Still not convinced?

- The financial investment – companies are willing to invest millions and millions of dollars into tools which might prevent a future critical bug from happening

- Hope for a brighter future?

Still not convinced?

- Hope for a brighter future?



Interactive verification tools

In this course, we will be using Dafny, a verification-aware programming language from Microsoft Research*

* now developed, funded, and widely used internally at Amazon

Why Dafny?

- It's modern (actively developed)

- It's used in real industry applications

- It can *cross-compile* to other languages: such as C#, Go, Python, Java, and JavaScript.

- It has a good IDE (VSCode extension)

Verification tools in other popular languages?

Yes!

SEE: Detailed list in lecture 6 README file)

Before we get started…

Note 1: Help on the project

Thing about what properties you want to verify

at compile time vs **at run time.**

I would love to see some projects that apply both tools successfully to different constraints!

Ex.: pre/postconditions vs. domain-specific constraints like a static analysis or Sudoku

If you’re not sure, come talk to me!

Note 2: Why cover theory?

Program verification is practical! Industry has invested millions and millions of $ into verifying software and hardware… (see these slides and many other examples in extras/verification-examples.md)

Course goals:

- to understand how verification works
- to apply verification to real-word projects

Why cover theory?

A: Verification is a lot of effort! From my experience, my best bet is you need a strong foundation in theory to understand and apply verification tools in practice.