# CS 318 Principles of Operating Systems

Fall 2021

# Lecture 22: System Reliability

**Prof. Ryan Huang**

JOHNS HOPKINS
WHITING SCHOOL
*of* ENGINEERING

# Teaser

## Civil Engineering

- Bridges don't fall

# Teaser

## Civil Engineering

- Bridges don't fall

## Mechanical Engineering

- Cars don't break

# Teaser

## Civil Engineering

- Bridges don't fall

## Mechanical Engineering

- Cars don't break

## Electrical Engineering

- City lights don't go off

# Teaser

## Civil Engineering

- Bridges don't fall

## Mechanical Engineering

- Cars don't break

## Electrical Engineering

- City lights don't go off

## Software Engineering

?

A problem has been detected and Windows has been shut down to prevent damage to your computer.

PFN_LIST_CORRUPT

If this is the first time you've seen this Stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup Options, and then select Safe Mode.

Technical information:
*** STOP: 0x0000004e (0x00000099, 0x00900009, 0x00000900, 0x00000900)

Beginning dump of physical memory
Physical memory dump complete.
Contact your system administrator or technical support group for further assistance.

# Why Is Reliable Software Hard?

## Technical factor

- Software is complex:
  - Exploding software state and set of possible behaviors
  - Hard to check all behaviors
- Execution environment contains nondeterminisms
- Construction approach is not rigorous

## Human factor

- To err is human
- Software requirements change
- Human beings use software in ways unexpected by designers

# Why Is Reliable Software Hard?

**A QA engineer walks into a bar.**

**He orders a beer** 

**He orders 0 beer**

**He orders 999999999 beers** 

**He order a lizard, -1 beer, a ueicbksjdhd** 

**First real customer walks in** 

- and asks where the bathroom is, the bar bursts into flames, killing everyone.

# What Is Software Reliability?

**Reliability is the probability that a software operates without failure in a given period of time in a specific environment**

- What is a failure?
- $Reliability = 1 - Probability(Failure)$
- Can be expressed as failure rate $\lambda$
- Mean Time Between Failure (MTBF, $1/\lambda$) is often reported
  - MTBF = 2000 hours => $\lambda$ = 0.0005/hour

**One important metric about software quality**

- Other metrics: efficiency, security, usability, maintainability, etc.

# Why Is Software Reliability Important?


**"Software is eating the world"**

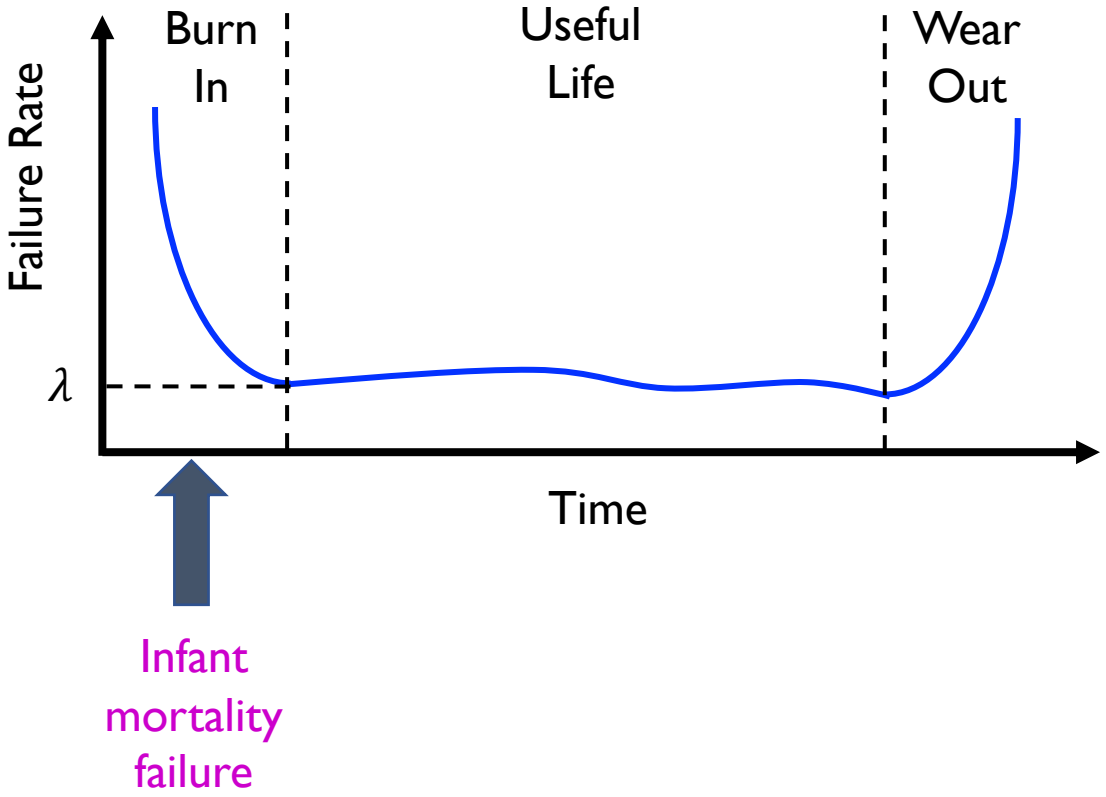
**Cost of software failure is high**

- Bugs in radiation-therapy Therac 25 caused tragedies of multiple deaths

# Software Reliability vs. Hardware Reliability

## The failure rate of a system usually depends on time

- Hard disk's failure rate in its fifth year > the rate in the first year




**The bathtub curve**

# Software Reliability vs. Hardware Reliability

## The failure rate of a system usually depends on time

- Hard disk's failure rate in its fifth year > the rate in the first year

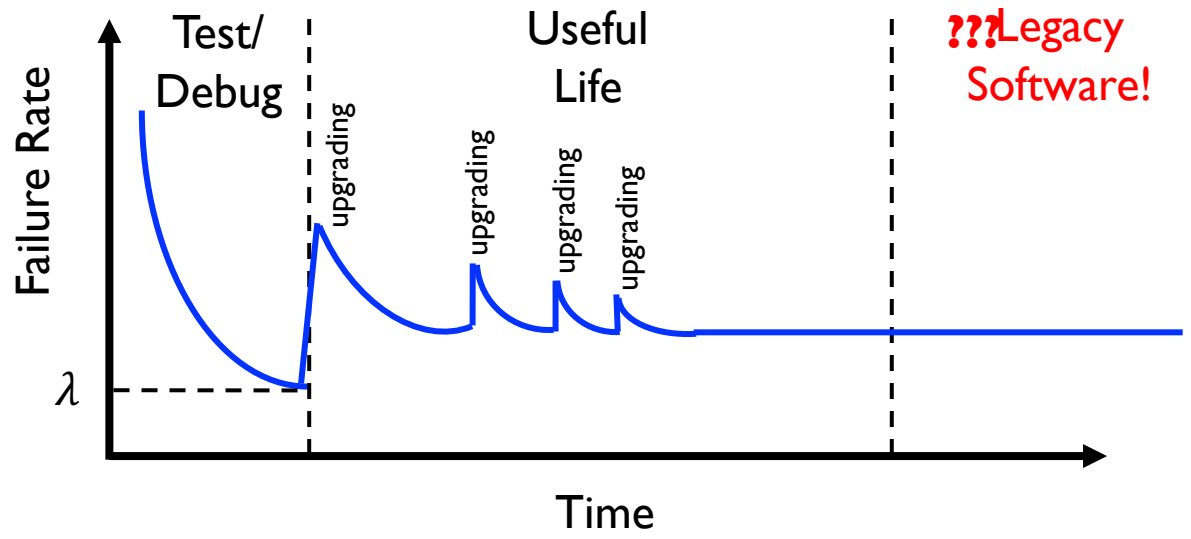
## Hardware typically exhibit the bathtub curve, but software don't

- Why?
- Hardware faults are mostly *physical faults*
- Software faults are *design/implementation faults*
  - Hard to visualize, classify, detect, and correct
  - Related to human factors, which we often don't understand well
- Software does not need "manufacturing"
  - Its quality does not change much once it's deployed

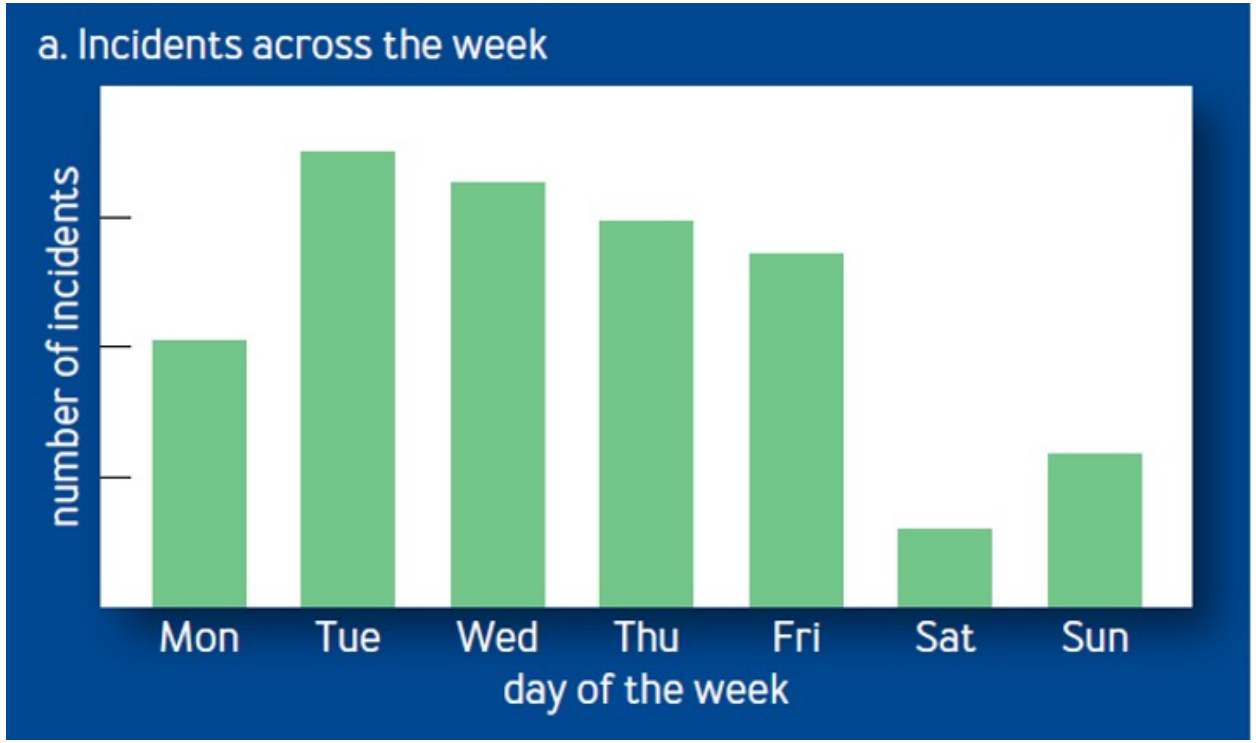
# What's The "Bathtub Curve" For Software?

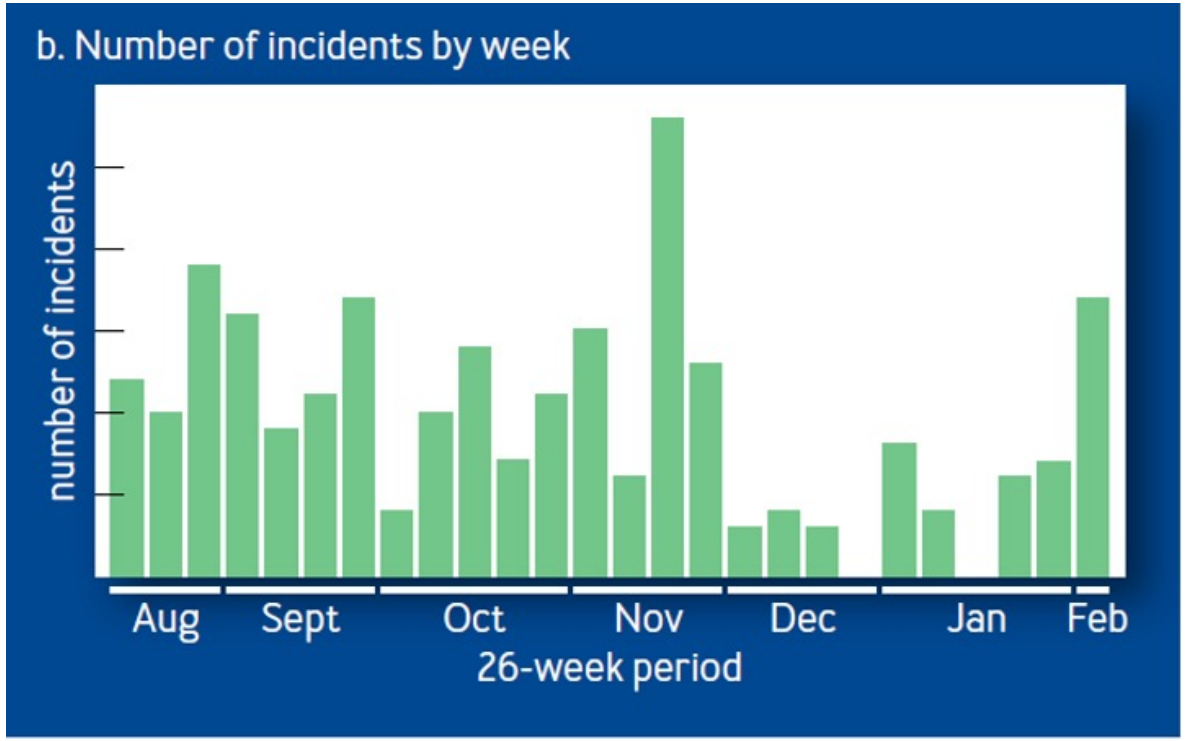
**What is the one major reason software fails?**

- Upgrades!



# Real-World System Failure Rates: Facebook




"Fail at Scale" [ACM Queue]

# Why Do Systems Fail?

## Hardware factors

- Power loss
- Disk wears out
- CPU random bit flip
- Memory corruption
- Room temperature too hot

## Software factors

- Bugs
- Configuration errors

## Human factors

- Human errors (e.g., `rm –rf /`)

# Why Do Systems Fail?

## A pioneer paper by Jim Gray:

- Study the commercial Tandem systems

| System Failure Mode | Probability | | MTBF in years |
|---|---|---|---|
| Administration | 42% | | 31 years |
| Maintenance: | 25% | | |
| Operations | 9% | (?) | |
| Configuration | 8% | | |
| Software | 25% | | 50 years |
| Application | 4% | (?) | |
| Vendor | 21% | | |
| Hardware | 18% | | 73 years |
| Central | 1% | | |
| Disc | 7% | | |
| Tape | 2% | | |
| Comm Controllers | 6% | | |
| Power supply | 2% | | |
| Environment | 14% | | 87 years |
| Power | 9% | (?) | |
| Communications | 3% | | |
| Facilities | 2% | | |
| Unknown | 3% | | |
| Total | 103% | | 11 years |

TANDEM COMPUTERS

**Why Do Computers Stop and What Can Be Done About It?**

Jim Gray

Technical Report 85.7
June 1985
PN87614

# Why Do Systems Fail?

## A pioneer paper by Jim Gray:

- Study the commercial Tandem systems
- Found that administration and software errors are the major contributors to failures
- Proposed software fault-tolerance techniques: process-pair and transactions


TANDEM COMPUTERS

Why Do Computers Stop and What Can Be Done About It?

Jim Gray

Technical Report 85.7
June 1985
PN87614

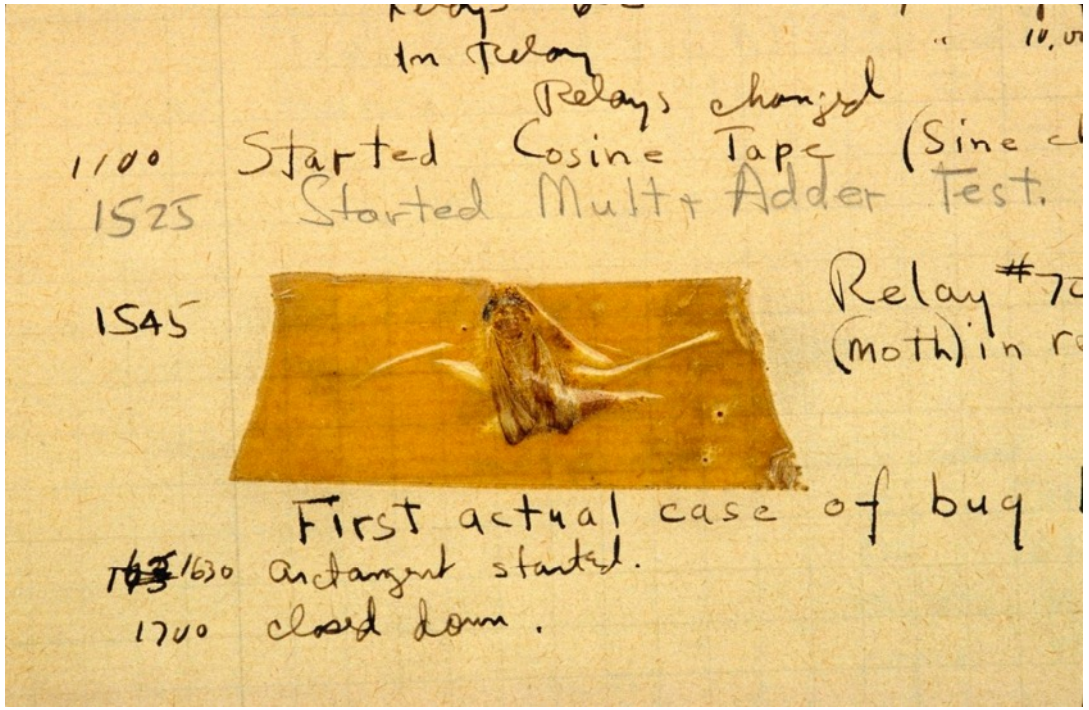
## Many papers followed up

- *Why do internet services fail, and what can be done about it?* [USITS '03]
- *Why Does a Cloud-Scale Service Fail Despite Fault-Tolerance?*

# Bugs

## The origin of "bug" is literally a bug

- Coined by U.S. Navy Admiral and computer science pioneer, Grace Hopper
- A moth got into a mechanical relay of Mark II supercomputer, jamming the system.



# Bugs in Programmers' Eyes

## Programmer's language translation guide

| What programmers say | What programmers mean |
|---|---|
| **Horrible hack** | Horrible hack that I didn't write |
| **Temporary workaround** | Horrible hack that I wrote |
| **It's broken** | There are bugs in your code |
| **It has a few issues** | There are bugs in my code |
| **Obscure** | Someone else's code doesn't have comments |
| **Self-documenting** | My code doesn't have comments |
| **I can read this Perl script** | I wrote this Perl script |
| **I can't read this Perl script** | I didn't write this Perl script |
| **Bad structure** | Someone else's code is badly organized |
| **Complex structure** | My code is badly organized |
| **Bug** | The absence of a feature I like |
| **Out of scope** | The absence of a feature I don't like |
| **Clean solution** | It works and I understand it |

| What programmers say | What programmers mean |
|---|---|
| **We need to rewrite it** | It works but I don't understand it |
| **Emacs is better than Vim** | It's too peaceful here, let's start a flame war |
| **Vim is better than Emacs** | It's too peaceful here, let's start a flame war |
| **IMHO** | You are wrong |
| **Legacy code** | It works but no one knows how |
| **^X^Cquit^\[ESC][ESC]^C** | I don't know how to quit Vim |
| **That can't be done** | It can be done, but it's boring and I don't want to do it |
| **No problem, people do this all the time. It's an easy fix.** | You might be the most idiotic person I've ever encountered |
| **Put that bug in the backlog with low priority** | Let's agree: nobody ever mention it again and ppl who do, will be shot |
| **These test environments are too brittle** | Works on my machine. Have you tried re-starting yours? |
| **Proof-of-Concept** | What I wrote |
| **Perfect solution** | How sales & marketing are promoting it |

# What Can Be Done About It?

## Bug detection

- Find bugs by analyzing source or binary code

## Testing

- Expose bugs by running software

## Failure isolation

- Mitigate damage of bugs at runtime

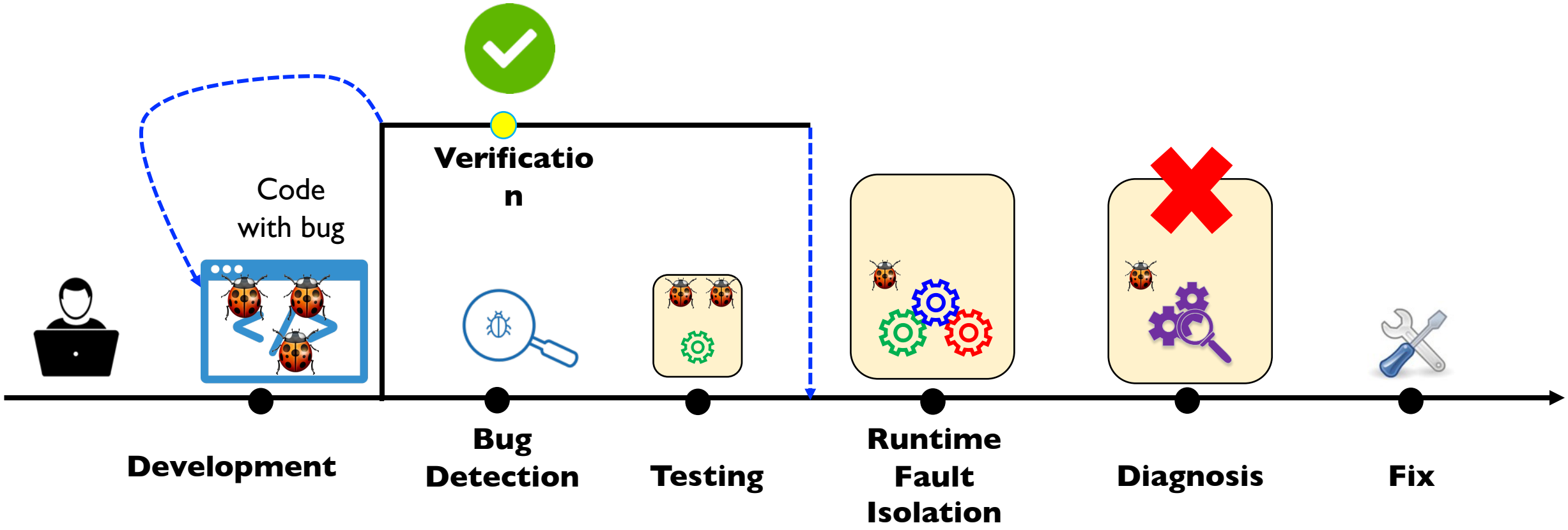
## Diagnosis

- Troubleshoot a bug after it has exhibited some symptom

## Fix

- Patch the software to remove the bug

# Life of A Bug and Reliability Efforts



# Bug Detection – Static Analysis

**Takes source code of a software, walk through the code flow structure, analyze program behavior, check rules**

- Some basic questions:
  - Where does the source of a variable come from
  - How does the value propagate through the function
  - What places use the value
- Different flavors: intra-procedural, inter-procedural, data-flow, control-flow, field-sensitive, etc.

**Relies on compiler techniques**

- Usually work on intermediate representation in static single assignment (SSA) form
- Popular tools: LLVM, Frama-C, Soot, FindBugs

# Bad News: No Silver Bullet

## No Perfect Static Analysis Method Exists

- Why?
- the general problem of finding all possible run-time errors in an arbitrary program is undecidable: reducible to the halting problem

## Each method makes trade-off between soundness and completeness

- A sound static analysis over-approximates the behaviors of the program
  - guaranteed to identify all violations
  - but may report false positives
- A complete static analysis under-approximates the behaviors of the program
  - every reported violation is a true violation
  - But no guarantee that all violations will be reported

# What Correctness Rule Should a System Obey?

## A hard question, depends on specific systems

- Often need domain specific knowledge and experience
- Rules are often undocumented or specified in ad hoc manner
- Manually discovering these rules is a daunting task
  - E.g., discovering such rules in Linux with millions of lines of code

## Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code [SOSP '01]

- Core insight: programmers have certain "beliefs" which are implied by the code they write.
  - `int a = *p;` ➔ `p` should be a non-null pointer
  - `unlock(l);` ➔ `l` was locked
- You can extract rules from the code rather than from programmers

# Bugs as Deviant Behavior (1)

**Some beliefs are "MUST" beliefs**

- p should be a non-null pointer

**Some beliefs are "MAY" beliefs**

- `spin_lock(l)` is followed by `spin_unlock(l)`, could be just a coincidence

**For "MUST" beliefs, any contradiction is an error**

**For "MAY" beliefs, an deviation is a *probable* error**

- Statistical approach to rank error probability
  - 999 out of 1000 times, `spin_lock(l)` is followed by `spin_unlock(l)` likely a true belief

**Key benefit: no prior knowledge of truth is required**

- If two beliefs contradict, one is an error, even though you don't know which one

# Bugs as Deviant Behavior (2)

## Define some generic rule templates

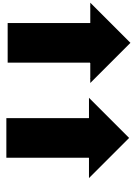
- <a> must be paired with <b>

## Keep a belief set for a program element

- Update belief set as the analysis proceeds

## Example:

```
/* Linux 2.4.1:drivers/isdn/avmb1/capidrv.c: */
if (card == NULL) {
  printk(KERN_ERR "capidrv-%d: ... %d!\n",
              card->contrnr, id);
}
```



**belief:** `card` is null

**belief:** `card` is non-null

**Contradiction!**

# Bugs as Deviant Behavior (2)

## Define some generic rule templates

- <a> must be paired with <b>

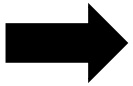
## Keep a belief set for a program element

- Update belief set as the analysis proceeds

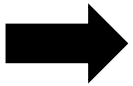
## Example:

```
/* Linux 2.4.7:drivers/char/mxser.c */

int mxser_write(struct tty_struct *tty, ...) {
  struct mxser_struct *info = tty->driver_data;
  unsigned long flags;
  if (!tty || !info->xmit_buf)
    return (0);
}
```



**belief:** `tty` is non-null



**belief:** `tty` is null

**Contradiction!**

# Bug Detection – Symbolic Execution

## Testing feeds a program with concrete data

- Downside: limited code coverage, some code paths are not explored

## Symbolic execution: feed a program with *symbolic* value

- Computation is based on symbolic values
- Output is expressed as a function of symbolic value
- Can generate a specific concrete input based on the symbolic expression
  - Testing input generation

# Symbolic Execution

**SE engine maintains a symbolic state $\delta$ and a symbolic path constraint $PC$**

- $\delta$ maps variable to symbolic expressions, initially empty
- $PC$ is a qualifier-free first-order formula over symbolic expression, initially true

**$\delta$ and PC are updated as program executes**

- at every assignment $v = e$, update $\delta$ by mapping $v$ to $\delta(e)$
- at every conditional statement `if (e) S1 else S2`, $PC$ is updated to $PC \wedge \delta(e)$ ("`then`" branch), and a new $PC' = PC \wedge \neg\delta(e)$ ("`else`" branch)
- execution continues along a path if the associated $PC$ is satisfiable

**Representative tool: KLEE**

- KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs [OSDI '08]

# Symbolic Execution Example

## Example code snippet to be symbolically executed

```
1   int twice (int v) {
2     return 2*v;
3   }
4   void testme (int x, int y) {
5     z = twice (y );
6     if (z == x) {
7       if (x > y+10)
8         ERROR;
9     }
10  }
11  int main() {
12    x = sym input();
13    y = sym input();
14    testme (x, y );
15  }
```

$\delta = \{x \mapsto x_0, y \mapsto y_0, z \mapsto 2y_0\}$

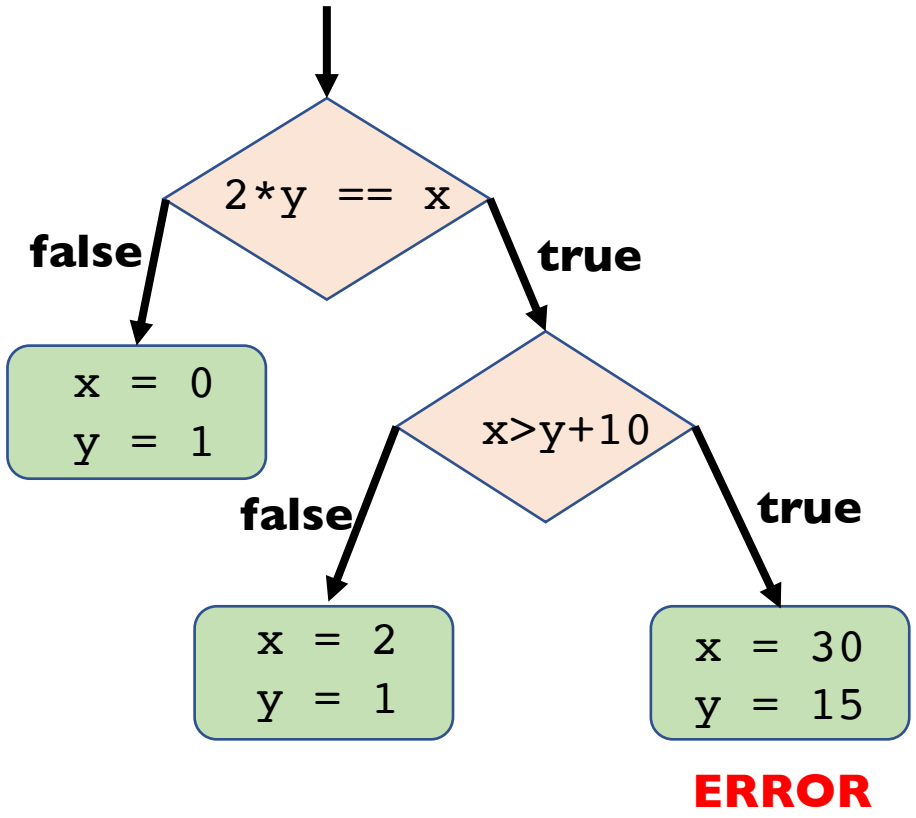
PC: $x_0 == 2y_0$ PC′: $x_0 \neq 2y_0$

PC″: $x_0 == 2y_0 \wedge x_0 > y_0 + 10$

$\delta = \{x \mapsto x_0\}$

$\delta = \{x \mapsto x_0, y \mapsto y_0\}$

$\delta = \{x \mapsto x_0, y \mapsto y_0\}$



# Debugging

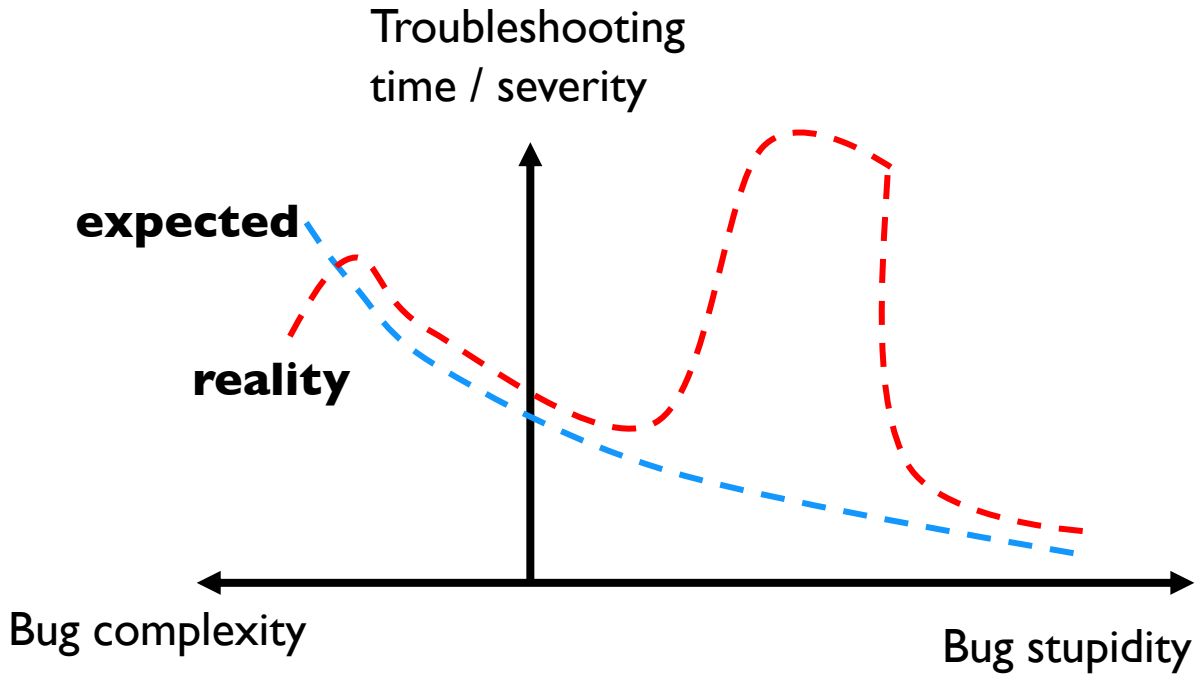
**"Debugging is like being the detective in a crime movie…**

**…where you are also the murderer"**

# Huang's Rule of Thumb on Debugging

**All bugs are obvious, *after* you debug them**

**Some bugs are "stupid", but stupid bugs ≠ easy bugs**



# Huang's Rule of Thumb on Debugging

**All bugs are obvious, *after* you debug them**

**Some bugs are "stupid", but stupid bugs ≠ easy bugs**

- After some point, the more time you spend on troubleshooting an issue, the more stupid the bug turns out to be
  - one-off error, `int` vs. `unsigned int`, `>` vs. `>=`
- Example:
  - https://azure.microsoft.com/en-us/blog/summary-of-windows-azure-service-disruption-on-feb-29th-2012/

**The more bugs you debug in a system, the deeper you understand about that system**

- Also why companies' new engineer training task is often debugging

# Debugging

**Ad-hoc: `printf`, systematic tool: `gdb`**

- examine program state, e.g., if a branch is taken, value of a variable
- compare the state with expected behavior
- if it deviates from the expected, how does it become like this

**Challenge 1: debugger may not be available**

- e.g., distributed system

**Challenge 2: hard to reproduce an issue in production**

- e.g., no core dump generated

**Challenge 3: root cause is far away from the failure site**

- e.g., why is this pointer becoming a null pointer?

# Logging: Source of Clues in Debugging

## Logging is an instrumental aid for debugging

- Often the only clues left in the crime scene (production environment)

## That's why the quality of logs is important

- Trade-offs among information, overhead, importance
- *Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold [SOSP '17]*
- *Be Conservative: Enhancing Failure Diagnosis with Proactive Logging* [OSDI '12]
- *Improving Software Diagnosability via Log Enhancement* [ASPLOS '11]

## Deducing information from logs is an art

- "The Science of Deduction"
- SherLog: Error Diagnosis by Connecting Clues from Run-time Logs [ASPLOS '10]

# Debugging In the Large

## How would Microsoft developers debug a Windows problem?

- OS is already deployed to customer computer
- Debug symbols not enabled at customer site
- Hard to convince customer to run a debugger

## Windows Error Reporting (WER) [Paper]

# Debugging In the Large

## How would Microsoft developers debug a Windows problem?

- OS is already deployed to customer computer
- Debug symbols not enabled at customer site
- Hard to convince customer to run a debugger

## Windows Error Reporting (WER) [Paper]

- A distributed system to collect Windows crash log from computers worldwide
- If a fix for the error exists, WER provides customer with the fix link
- WER aggregates error reports and performs automatic diagnosis if possible

# Other Interesting Topics

## Bug fixing

- Bug fixes can become bug again
  - The fixes are only workaround or the other parts of software changes

## System verification

- Passing testing and static analysis tools does not mean the software is bug-free
- How can we *prove* that a software is correct *under all circumstances*

## Configuration errors

- Not just code bug or hardware issue, human error!

## Failure detection

- Production software does not always simply crash, often exhibit gray failure

## Failure isolation

## Fault tolerance

# If You Are Interested In Knowing More…

https://www.cs.jhu.edu/~huang/pubs.html

**We should talk ☺**

# Next Time…

**Final review**