Why Do We Need to Engineer Systems?
A Look at History's Greatest Computer System Failures

ECOR 1055: Introduction to Engineering Disciplines I

Dr. Mohamed Ibnkahla
(Slides mostly prepared by Dr. Jason Jaskolka)

Department of Systems and Computer Engineering
Faculty of Engineering and Design
Carleton University Ottawa, ON, Canada

Outline

Lecture Objectives

1. Know and understand the need for computer systems engineering as a discipline
2. Recall some of history's greatest computer system failures and their causes

What is Engineering?

Engineering [Professional Engineers Act 1990]

Any act of planning, designing, composing, evaluating, advising, reporting, directing or supervising that requires the application of engineering principles and concerns the safeguarding of life, health, property, economic interests, the public welfare or the environment, or the managing of any such act

Computer Systems Engineering

Computer Systems Engineering

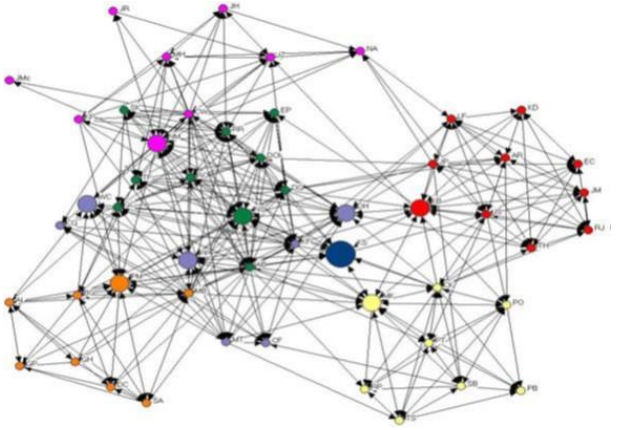
Combines hardware and software to design and implement integrated computer systems for applications in such areas as robotics, artificial intelligence, aerospace and avionic systems, multimedia applications and cloud computing.

- Computer systems engineers develop, test, and evaluate integrated software and hardware by combining their knowledge of engineering, computer science, and mathematical analysis

Reasons for Systems Failure

- Many "system failures" are a consequence of two factors:

1. *Increasing system complexity*



2. *Failure to use appropriate engineering methods*

The Importance of Computer Systems Engineering

- Computer systems are pervasive in our lives
  - This includes critical systems that affect our well-being
- More and more, individuals and society rely on advanced software/hardware systems
  - Reliable and trustworthy systems must be produced economically and quickly
- Many reported stories of poor engineering practices leading to catastrophes

Mariner 1 Rocket (1962)

Mariner 1 Rocket (1962)

Failure

Mariner 1 was the first spacecraft of the American Mariner program, designed for a planetary flyby of Venus. It cost $18.5M in 1962. It was launched aboard an Atlas-Agena rocket on July 22, 1962. Shortly after takeoff the rocket responded improperly to commands from the guidance systems on the ground, setting the stage for an apparent software-related guidance system failure. With the craft effectively uncontrolled, a range safety officer ordered its destructive about 294.5 seconds after launch.

Mariner 1 Rocket (1962)

Cause

The Mariner 1 Post Flight Review Board determined that the omission of a hyphen (overbar) in coded computer instructions in the data-editing program allowed transmission of incorrect guidance signals to the spacecraft. During the periods the airborne beacon was inoperative the omission of the hyphen in the data-editing program caused the computer to incorrectly accept the sweep frequency of the ground receiver as it sought the vehicle beacon signal and combined this data with the tracking data sent to the remaining guidance computation. This caused the computer to swing automatically into a series of unnecessary course corrections with erroneous steering commands which finally threw the spacecraft off course.

Ariane 5 Flight 501 (1996)

Ariane 5 Flight 501 (1996)

Failure

On June 4, 1996, on its maiden flight, the Ariane 5 was launched and performed perfectly for 40 seconds. Then, it began to veer off course. After detection of loss of integrity, the rocket self-destroyed. The rocket was lost as well as the four satellites it contained. The total cost was $500M and may be much more in reputation.

Ariane 5 Flight 501 (1996)

Cause

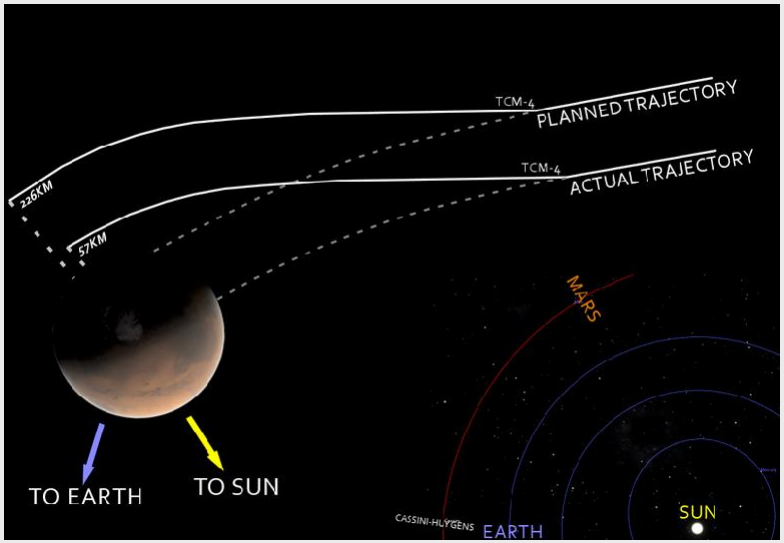
One subsystem of the flight program, the SRI (the inertial reference system, which provides attitude data about the rocket), was reused from Ariane 4. Reuse was intended to reduce risk, increase productivity, and quality. However, this reused software contained a bug that was not exploited in the Ariane 4. The problem was that the Ariane 5 had a different preparation sequence that exposed the bug. This is an example of poor specifications, usage testing, and exception handling.

Mars Climate Orbiter (1999)

Mars Climate Orbiter (1999)

Failure

The Mars Climate Orbiter probe was launched on December 11, 1998 aboard a Delta II 7425 launch vehicle. Mars Climate Orbiter began the planned orbital insertion maneuver on September 23, 1999, however, it went out of radio contact when the spacecraft passed behind Mars 49 seconds earlier than expected, and communication was never reestablished. The spacecraft encountered Mars at a lower than anticipated altitude and it was either destroyed in the atmosphere or re-entered space after leaving Mars' atmosphere.



Mars Climate Orbiter (1999)

Cause

The loss of the $125M USD spaceraft was reportedly caused by a discrepancy between the calculated and measured position of the spacecraft. It turns out that one team, contrary to the requirements specification, programmed a key software system to calculate in units of *pound-seconds* while another team expected the calculations to be measured in *newton-seconds* (as specified in the requirements). This resulted in incorrect predictions of the actual position of the spaceraft.

Y2K (1999/2000)

Y2K (1999/2000)

Failure

The Y2K bug was a computer flaw that sparked havoc and problems when dealing with dates beyond December 31, 1999. The flaw, faced by computer programmers and users all over the world on January 1, 2000, is also known as the "millennium bug." Many skeptics believe it was barely a problem at all, but businesses spent billions on programmers to fix a glitch in legacy software. While no significant computer failures occurred, preparation for the Y2K bug had a significant cost (estimated at $500B) and time impact on all industries that use computer technology.

Y2K (1999/2000)

Cause

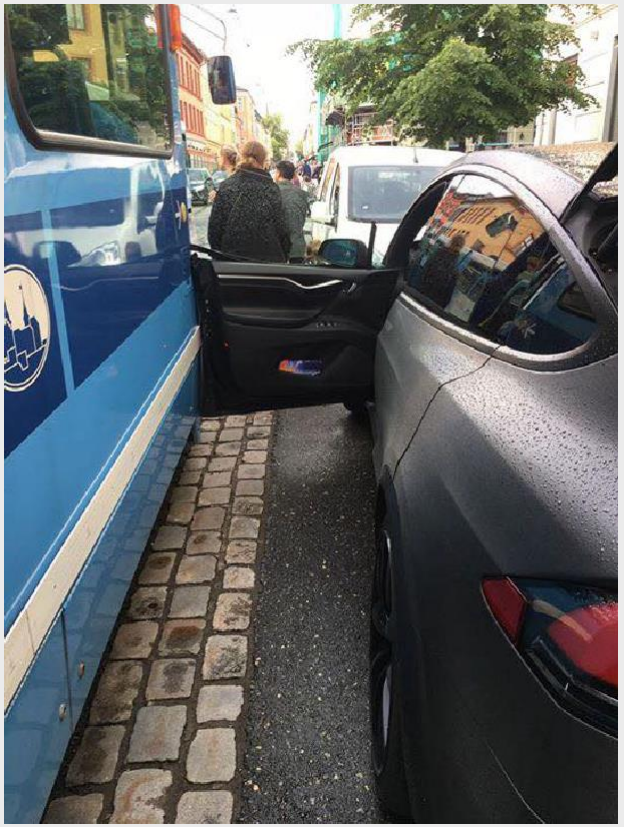
To save computer storage space, legacy software often stored the year for dates as two digit numbers, such as “99” for 1999. The software also interpreted “00” to mean 1900 rather than 2000, so when the year 2000 came along, there was widespread fear that systems would cease to function correctly.

Tesla Model X Self-Presenting Doors (2017)

Tesla Model X Self-Presenting Doors (2017)

Failure

The front doors on the Tesla Model X have electric power and can open on their own. The problem with this feature is that the doors were opening when they actually had no business doing so. One case was in Norway, where the driver's door opened automatically right in the path of a passing streetcar while he was putting something in the trunk. The public transit vehicle only sustained minor damage. Hitting something like a bus or a streetcar is one thing, but having a door open in front of a cyclist can lead to much more serious consequences.



Tesla Model X Self-Presenting Doors (2017)

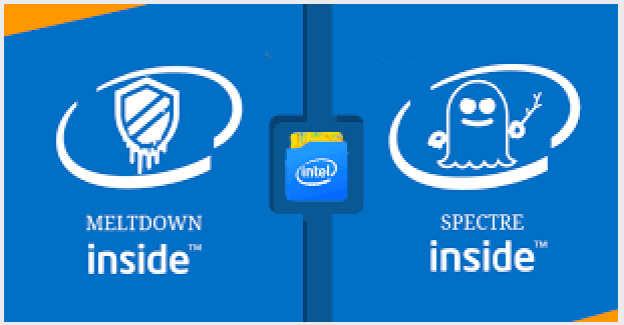
Cause

It is unclear, but there is widespread speculation that a software bug was causing the vehicle to believe that the key fob was much closer to the door than it actually was. Reliance on ultrasonic sensors as proximity sensors, combined with a failed obstacle avoidance system within the door is a likely culprit. An incorrect over-the-air update is also thought to have contributed to the introduction of such a bug. Others believe that the cause was human error.

Spectre and Meltdown (2018)

Spectre and Meltdown (2018)

Failure



Spectre and Meltdown are vulnerabilities present in nearly all computers. While Meltdown is only effective against Intel CPUs (1995-2018), as well as some ARM processors, Spectre affects practically every single computer system as of 2019. Meltdown exploits the way modern CPUs function and allows processes running on a computer to see all information currently being used by the CPU by avoiding security measures designed to stop this. The implications of this are terrible—attackers could see passwords, sensitive financial information, images, and practically anything they wanted without users' knowledge, all while avoiding antivirus software. Spectre is similar in nature to Meltdown and works by tricking a program into accessing innocent-seeming memory but actually allowing an attacker to read this data and potentially retrieve sensitive information.

Spectre and Meltdown (2018)

Cause

Both Meltdown and Spectre are caused by the widespread use of a technique called *speculative execution* in which processors eagerly and proactively execute instructions even before they are actually needed by the program. The speculatively computed material is then faster, but the primary discovery of Meltdown and Spectre was that it is insufficiently secured, and thus provides a way to leak sensitive information.
The only real fix for Meltdown is to eventually physically replace all the chips, a change which will take at least a full hardware generation to propagate. Spectre is more sophisticated, and may have no real fix at all.

Uber Self-Driving Car (2018)

Uber Self-Driving Car (2018)

Failure

In March 2018 in Tempe, AZ, an Uber self-driving vehicle fatally struck a pedestrian walking a bicycle. The NTSB released over 400 pages ahead of a November 2019 meeting aimed at determining the official cause of the accident and reporting on its conclusions. The NTSB's technical review of Uber's autonomous vehicle technology revealed a cascade of poor design decisions that led to the car being unable to properly process and respond to the pedestrian's presence as she crossed the roadway with her bicycle.

Uber Self-Driving Car (2018)

Cause

A radar on the vehicle first detected the pedestrian roughly six seconds before impact, followed quickly by the Lidar. However, the car's self-driving system did not have the capability to classify an object as a pedestrian unless they were near a crosswalk. For the next five seconds, the system alternated between classifying the pedestrian as a vehicle, a bike, and an unknown object. When the vehicle thought the pedestrian was a vehicle or bicycle, it assumed she would be travelling in the same direction as the Uber vehicle but in the neighbouring lane. When it classified her as an unknown object, it assumed she was static. Worse still, each time the classification flipped, the car treated her as a brand new object. That meant it could not track her previous trajectory and calculate that a collision was likely, and thus did not even slow down. Tragically, the vehicle's automatic braking system had been disabled because its radars could have interfered with Uber's self-driving sensors.

Boeing 737 MAX 8 (2019)

Boeing 737 MAX 8 (2019)

Failure

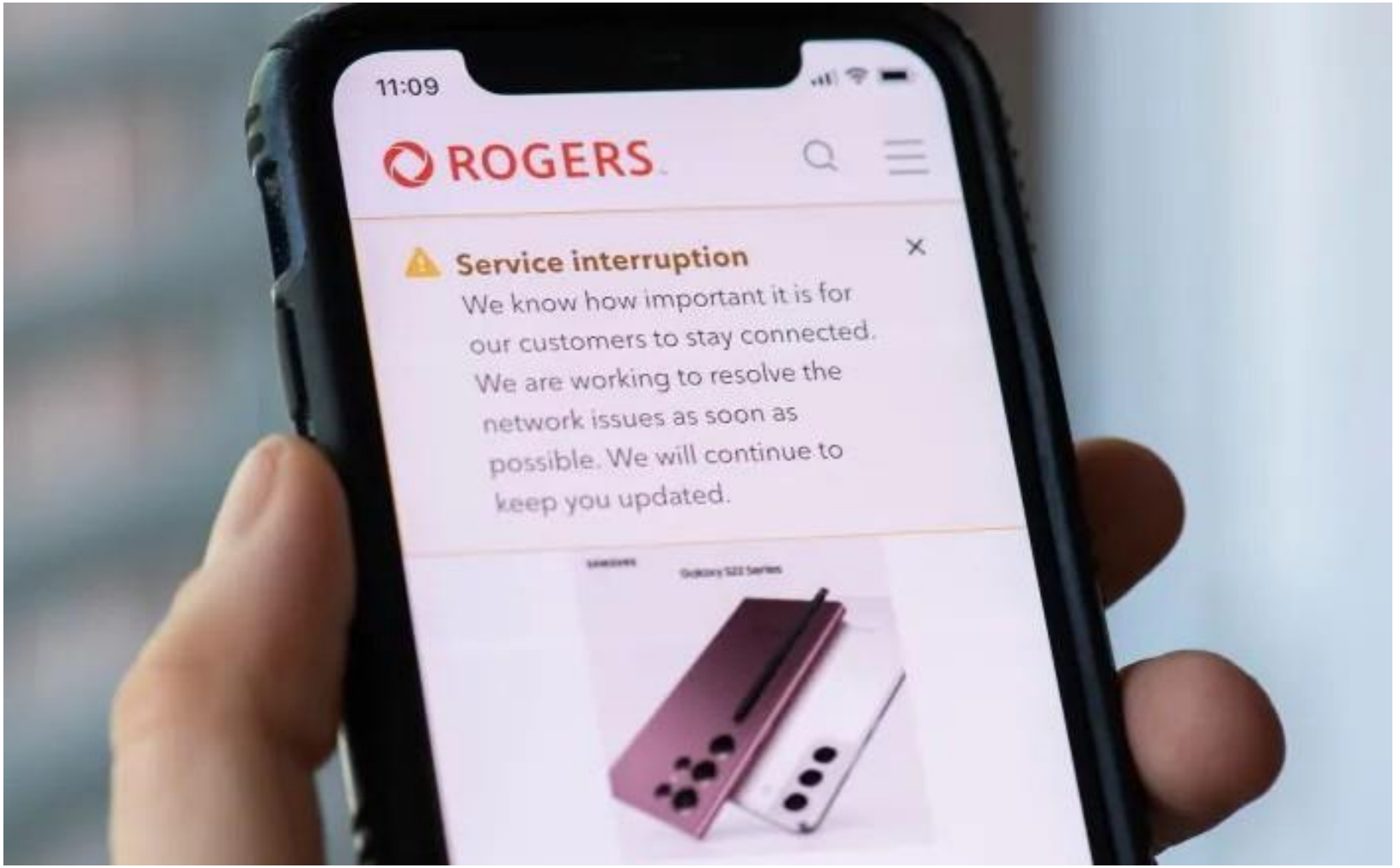
Almost as soon as the wheels of Ethiopian Airlines Flight 302 spun free from the runway on March 10, 2019, the digital displays for altitude, airspeed and other basic information went haywire. The plane was climbing too steeply and was in imminent danger of falling from the sky. Soon, a cascade of warning tones and coloured lights and mechanical voices filled the cockpit. Another powerful automated flight-control system called the MCAS abruptly pushed down the jet's nose. The pilots wrestled with the controls, desperate to raise the nose of the plane. Flight 302 nose-dived at nearly the speed of sound killing all 157 people aboard. Five months earlier, Lion Air Flight 610 had plunged into the Java Sea, killing 189 people, under similar circumstances.

Boeing 737 MAX 8 (2019)

Cause

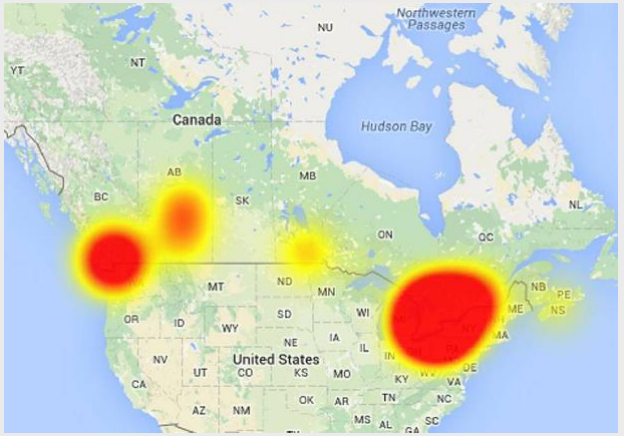
Four minutes into the flight, erroneous signals from a malfunctioning sensor tricked the onboard computers into believing the nose was angled too high, causing the Maneuvering Characteristics Augmentation System (MCAS) to push it down again and again. Regulators have since blamed the crashes on the MCAS, its reliance on a single sensor, and Boeing's decision not to tell pilots about the new system. At the root of the miscalculations, though, were Boeing's overly optimistic assumptions about pilot behaviour. In designing the flight controls for the 737 MAX, Boeing assumed that pilots trained on existing safety procedures should be able to sift through the jumble of contradictory warnings and take the proper action 100% of the time within four seconds.

Rogers Outage (2022)



Rogers Outage (2022)

Failure



Millions of Canadians woke up on the morning of July 8, 2022 to find they had no internet or mobile service. Debit transactions at stores failed. E-transfers didn't go through. Canadians couldn't reach 9-1-1. Government services reported disruptions because phone lines were down. Estimates indicate that around 25% of Canada lost Internet connectivity. The outage lasted nearly an entire day. The total cost of the outage is difficult to estimate and will not likely be know for some time, but Rogers will pay an estimated $150 million in credits to customers following the outage. Rogers also plans on separating its wireless and wireline services to prevent one outage from taking down both networks in the future.

Rogers Outage (2022)

Cause

While the cause of the outage is debated, Rogers CEO Tony Staffieri provided the following explanation of the outage:

"We now believe we've narrowed the cause to a network system failure following a maintenance update in our core network, which caused some of our routers to malfunction early Friday morning . . . We disconnected the specific equipment and redirected traffic, which allowed our network and services to come back online over time as we managed traffic volumes returning to normal levels."

Lecture Summary

- Systems can fail for many reasons
  - Poor practices
  - Wrong assumptions
  - Overconfidence
  - Coding errors
  - Failing to anticipate changes
  - Failing to follow specifications
  - Inadequate testing
  - etc.
- Computer systems engineering helps to define processes and practices to manage the increasing complexity of integrated software/hardware systems
- Without proper engineering practices, we can face systems that can fail in catastrophic and spectacular ways
  - Professional engineers have a clearly defined duty to society, which is to regard the duty to public welfare as paramount [Professional Engineers Act 1990]

References & Further Reading

Government of Ontario
Professional Engineers Act, R.S.O. 1990, c. P.28
Available: https://www.ontario.ca/laws/statute/90p28#BK0

U.S. Nuclear Regulatory Commission
Digital Systems Software Requirements Guidelines – Failure Descriptions
NUREG/CR-6734, Vol. 2, June 2001
Available: https://www.nrc.gov/docs/ML0123/ML012330184.pdf

Questions?