

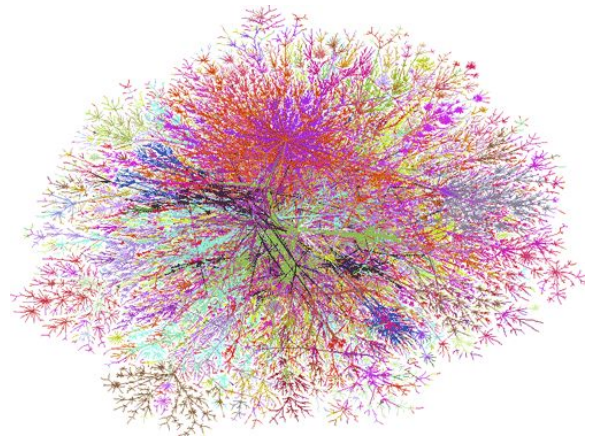
# Computer Science Module 1: Reliability & Liability

Prepared by Dr. Daniel Rosiak, Prof. Shannon E. French, and Beth Trecasa – June 2021

# Introduction

It is difficult to overstate how dependent we are today on computers and computerized systems to facilitate so many of our daily activities. Such systems now govern most of modern communication, transportation, finance, retail, healthcare, military systems, and more.

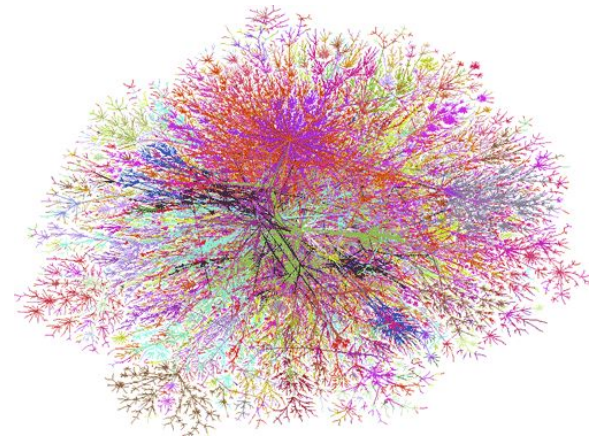
When computerized systems work correctly, they can of course save us time, money, and enable the accomplishment of many other activities. But when they fail or something goes wrong, any of those benefits can quickly be overturned by rather disruptive or impactful harms. Failures of computer-integrated systems can ramify throughout many areas of society, resulting in lost time, lost money, social injustice, and even injury or loss of life.



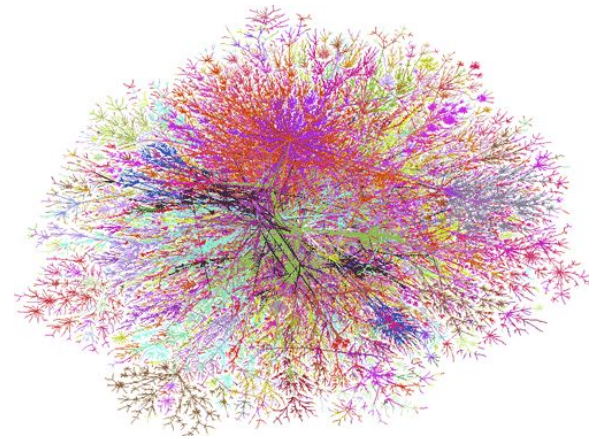
# Introduction

Computers and code nearly always form part of larger systems – like financial systems, health-care systems, transportation systems, etc. – and fundamentally it is the reliability of the *entire system* that is most important.

A system that is designed well is one that can tolerate the malfunction of any single component without failing or causing harm.



# Introduction



These two-paired modules are designed to familiarize you with:

- some of the ways in which computerized systems have proven to be unreliable, what we can do to make them more reliable,
- what the broader ethical stakes are, and
- general practices we can implement to encourage greater awareness and anticipation of the risks.

# Specific Objectives

The main purpose of this two-part Reliability & Liability module is to invite aspiring computer scientists and others involved in the construction, design, upkeep, and testing of large systems or components thereof to increase their awareness of some of the broader ethical risks associated with the variety of ways things can go wrong.

## Module 1

- A number of carefully-selected **examples** to illustrate **three main errors types**
- Important **lessons** drawn from each of those examples
- An **assignment** for students

## Module 2

- **General tools** to better identify and mitigate ethical risks
- An **assignment** for students
- Further **readings and resources**

# Three Error Types

We can identify three main sorts of errors, each of which will be illustrated in turn via concrete examples and discussion of those examples:

1. Errors in **data-entry** or **data-retrieval**
2. Errors, bugs, or code features that enable **system malfunction**
3. Errors, bugs, or code features that enable **system failure**

# 1. Errors in data-entry or data-retrieval

**Example 1: National Crime Information Center (NCIC)**

**Example 2: Amazon AWS Crash**

**Example 3: Hawaii Missile Alert**

# 1. Errors of data entry or retrieval

First, let us look at examples where the user or the computer-human interface turns out to be the weak link in the system, leading to some problem. We can call these issues **system errors due to data entry or data retrieval**.

This is just what it sounds like. Sometimes, computerized systems fail or behave in unexpected ways as a result of wrong data having been entered into them or because people incorrectly interpret the data they retrieve.





# 1. Errors of data entry or retrieval

While it is tempting to want to focus on a particular mistake made by an individual person entering or retrieving the data, we should keep in mind that any system

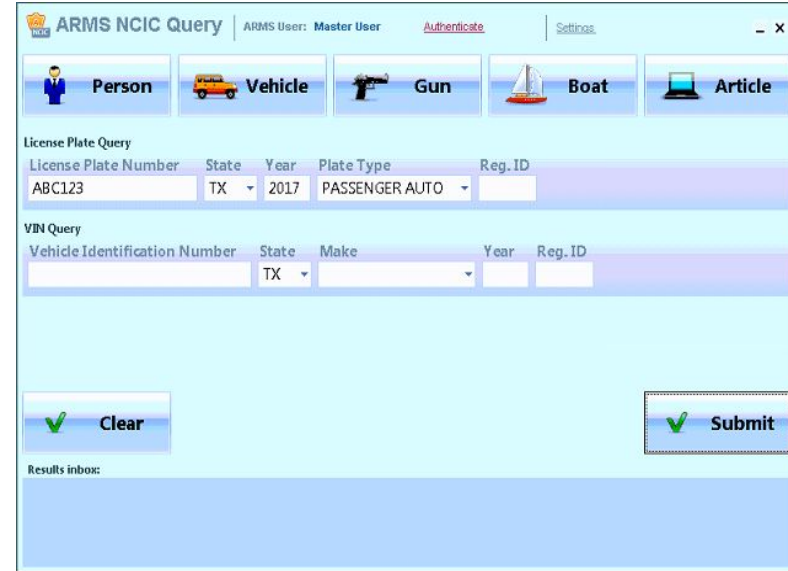
- like the *voting system* failing after incorrect records inputted into the computer database disqualified thousands of voters, or
- like the *criminal justice system* failing when a person is arrested after being confused with another person in a database

is larger than the individual person or persons who make the error.

Therefore, it is often more useful to focus on how such local errors are possible in the first place, how they are allowed to propagate through a system to become global or larger-impact problems, and how to anticipate and mitigate the risks associated with this.

# Example 1: National Crime Information Center (NCIC)

The FBI National Crime Information Center (NCIC) is a computerized database of criminal justice information made available to federal, state, and local law enforcement and other criminal justice agencies for ready access by the criminal justice agency making an inquiry. This information is designed to assist authorized agencies law enforcement objectives, such as apprehending fugitives, locating missing persons, looking up criminal records, locating and returning stolen property, etc.

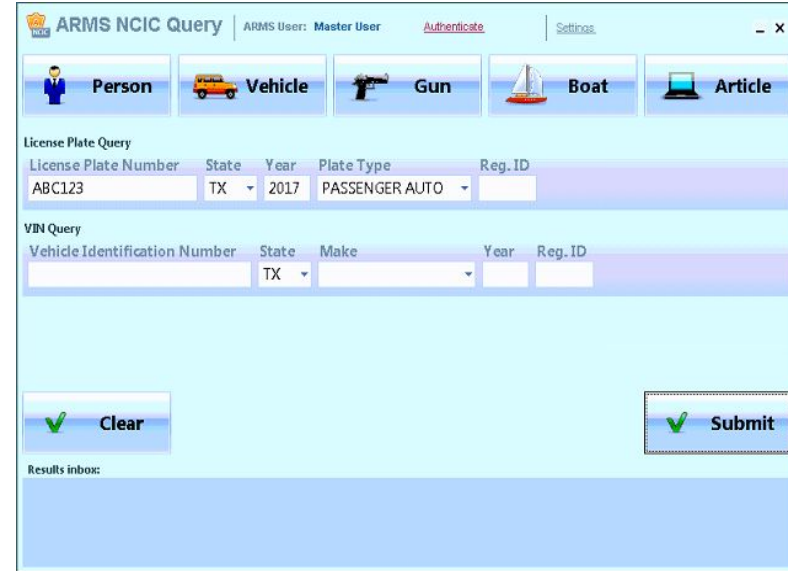


The screenshot displays the ARMS NCIC Query interface. At the top, it shows the user is logged in as 'Master User' and provides an 'Authenticate' link and a 'Settings' button. Below this is a navigation bar with five tabs: 'Person' (with a person icon), 'Vehicle' (with a bus icon), 'Gun' (with a handgun icon), 'Boat' (with a sailboat icon), and 'Article' (with a laptop icon). The 'License Plate Query' section contains a table with columns: License Plate Number (text input with 'ABC123'), State (dropdown menu with 'TX' selected), Year (text input with '2017'), Plate Type (dropdown menu with 'PASSENGER AUTO' selected), and Reg. ID (text input). Below this is the 'VIN Query' section with columns: Vehicle Identification Number (text input), State (dropdown menu with 'TX' selected), Make (dropdown menu), Year (text input), and Reg. ID (text input). At the bottom of the form area, there are two buttons: 'Clear' (with a green checkmark icon) and 'Submit' (with a green checkmark icon). Below the buttons is a 'Results inbox:' label followed by a large empty blue area.

# Example 1: National Crime Information Center (NCIC)

Tens of thousands of law enforcement agencies have access to these data files, and the NCIC processes more than 13 million requests for information each day.

For example, a police officer may initiate an NCIC search during a traffic stop to find out if the vehicle is stolen or there is a warrant out for the driver, and the system supplies records and answers to such queries.



The screenshot displays the ARMS NCIC Query interface. At the top, it shows the user is logged in as 'Master User' and provides options to 'Authenticate' and 'Settings'. Below this, there are five main search categories: Person, Vehicle, Gun, Boat, and Article, each with a representative icon. The 'License Plate Query' section includes fields for License Plate Number (containing 'ABC123'), State (a dropdown menu set to 'TX'), Year (containing '2017'), Plate Type (a dropdown menu set to 'PASSENGER AUTO'), and Reg. ID. The 'VIN Query' section includes fields for Vehicle Identification Number, State (a dropdown menu set to 'TX'), Make, Year, and Reg. ID. At the bottom of the form, there are two buttons: 'Clear' and 'Submit', both featuring a green checkmark icon. Below the buttons is a 'Results inbox:' label and a large empty blue area for displaying search results.

# Example 1: National Crime Information Center (NCIC)

A number of critics of the NCIC have pointed out ways in which the NCIC has led to a variety of injustices and privacy violations of innocent people, such as:

- Erroneous records entered in the database can lead law enforcement agencies to arrest innocent persons
- Typographical errors made by law enforcement checking the database have led to false arrests<sup>1</sup>
- Innocent people with the same name as that of individuals listed in the arrest warrants database have been mistakenly arrested<sup>2</sup>
- Corrupt law enforcement employees with access to NCIC have sold information, altered, deleted, and otherwise misused records<sup>3</sup>

# Example 1: National Crime Information Center (NCIC)

For concrete instances of the many stories of police making false arrests based on information they retrieved from the NCIC, here are two:

1. Roberto Perales Hernandez was jailed twice in three years as a suspect in a Chicago residential burglary, even though he had never been to Chicago in his life. The authorities had confused him with another Roberto Hernandez due to a single entry in the NCIC. The two Roberto Hernandezes were the same height, about the same weight, had brown hair, brown eyes, tattoos on their left arms, shared the same birthday, and had Social Security numbers that differed by just one digit.
2. Terry Dean Rogan was arrested five times – and twice at gunpoint! – for crimes he didn't commit. The NCIC had erroneously listed him as wanted for murder and robbery, even after the actual suspect using his name had been identified.

## Example 2: Amazon AWS Crash:

*“Your internet will return in 2-5 business days!”*

On Feb. 28, 2017, the Amazon AWS service crashed, causing many websites hosted by the software to become unresponsive. Amazon conducted an internal investigation and concluded that during a simple debugging, a single service member had executed a single command intended to remove a negligible amount of servers to help speed up the process.

While Amazon was able to restore service fairly quickly and no data was permanently lost, it was reported that S&P 500 companies lost an estimated \$150 million, and U.S. financial services companies lost even more during the outage. All this economic carnage from entering a simple command incorrectly!

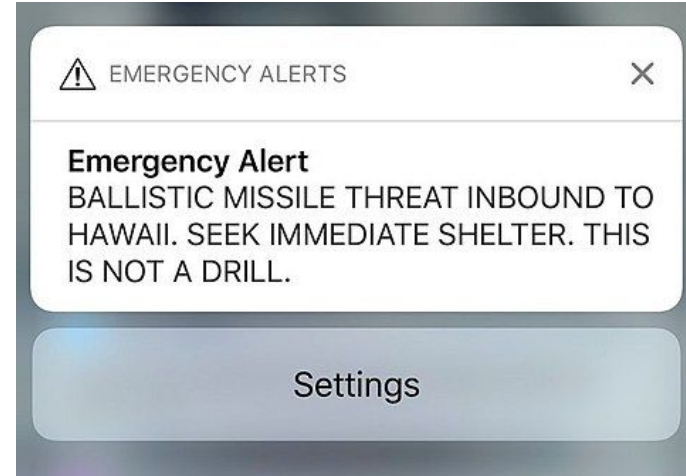


- Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region: <https://aws.amazon.com/message/41926/>
- <https://www.wsj.com/articles/amazon-finds-the-cause-of-its-aws-outage-a-typo-1488490506>

# Example 3: Hawaii Missile Alert

Hawaii is the only state with a pre-programmed Wireless Emergency Alert that can be sent directly to wireless devices if a ballistic missile is heading toward the state.

This is partly because if a missile were ever fired from North Korea, the missile would take approximately just 20 minutes to reach Hawaii.

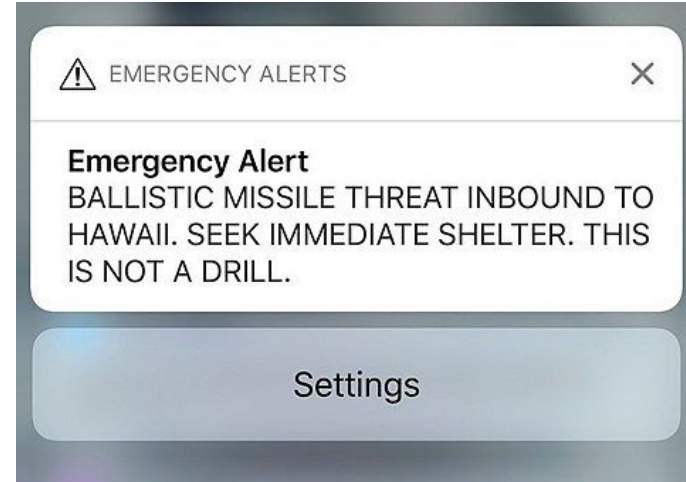


# Example 3: Hawaii Missile Alert

In 2018, the alert had been inadvertently sent out by an employee of the Emergency Management Agency during a shift change.

During the shift change, a supervisor initiated an unscheduled drill in which he contacted emergency management workers in the guise of an officer from US Pacific Command.

The supervisor deviated from the script, at one point saying "This is not a drill," although he did state both before and after the message, "Exercise, exercise, exercise," to indicate that it was in fact a test.

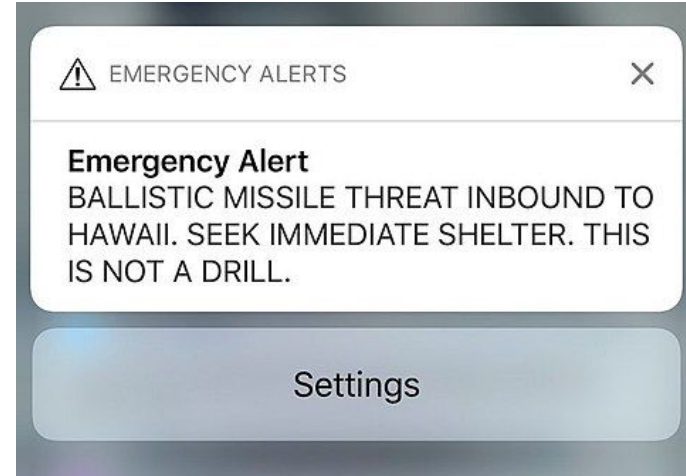




# Example 3: Hawaii Missile Alert

Upon hearing the supervisor's statement, the employee assuming their post believed there was an actual emergency, and proceeded to click the alert button from a dropdown of options, which would send out an actual notification on Hawaii's emergency alert interface.

The employee then clicked through a second screen, which had been intended as a safeguard, to confirm.



# Example 3: Hawaii Missile Alert

Hawaii Emergency Management Agency officials were subsequently asked for a screenshot of the interface the employee was looking at when the false alert was sent out. They gave a “facsimile” of the UI (not the actual screen, for security purposes), the archaic design, logic and layout of which was subsequently widely criticized.

## “DRILL-PACOM (DEMO) STATE ONLY”

was the link the employee should have clicked on for the test.

Listed further below is the link “**PACOM (CDW) — STATE ONLY,**” the link that he did in fact click, leading to the incoming ballistic missile alert sent to residents and visitors statewide.

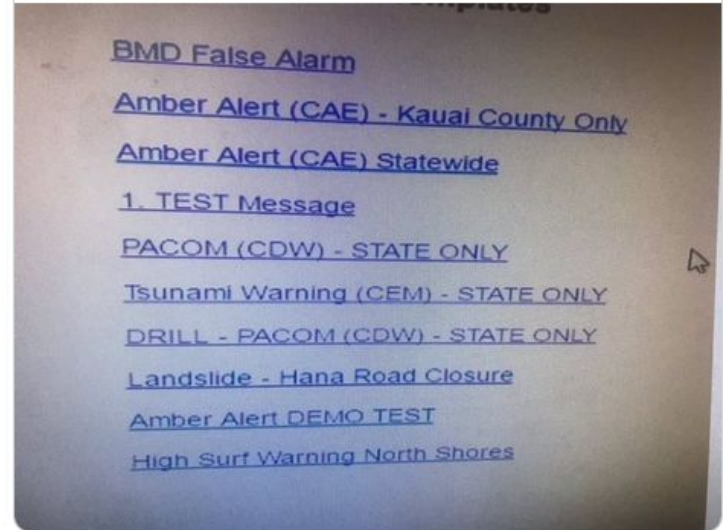


This is not the kind of interface you would expect to see for something this important.



Replying to @CivilBeat

The BMD False Alarm link is the added feature to prevent further mistakes #Hawaii



5:40 AM · Jan 16, 2018



# Lessons Learned: Errors of data entry or retrieval

From the perspective of those responsible for well-engineered systems and the code-elements of such systems, it may appear (and may be the case) that certain aspects of such “user errors” are inevitable.

The proper response to these things may indeed largely depend on the proper training of human users. But as ethically responsible coders, there are still general principles we can adopt that will allow us to do all we can to avert such errors or at least minimize the harmful effects of such unfortunate cases.

- Norman, Don. The design of everyday things. ISBN 978-0-465-06710-7.

# Lessons Learned: Errors of data entry or retrieval

Adopting a modified version of an insight from Don Norman\*:

- *Don't think of the user as making errors; think of the actions as approximations of what the system intends.*
- *Anticipate and build into the system tolerances and safeguards to the potential “errors” in such approximations.*
- *Remember that a well-engineered system does not fail when a single component fails. If there are ways a single human user can cause large-scale deviations from intended use, the effects of such local deviations on the broader system should be anticipated, minimized, and safeguarded against as much as possible.*

# Back to the Errors: Two Errors of Another Sort

Errors of data-entry or retrieval can be harmful. However, it remains the case that their analysis is typically comparatively uncomplicated, and the measures we can take are often straightforward.

Even supposing data entered into, or retrieved from, a computer system are correct, the system may still produce wrong or undesirable results, and may even collapse entirely if the errors are serious enough. In such cases, we can identify the other two main sorts of errors:

**2. Errors, bugs, or code features enabling system malfunction**

**3. Errors, bugs, or code features enabling system failure**

The ethical risks and liabilities of these two sorts of errors are generally harder to mitigate and/or involve the greatest potential for harm.

## **2. Errors/Bugs enabling system malfunction**

**Example 1: Speel chek**

**Example 2: High-Risk Inmates Mistakenly Released**

**Example 3: Rent Miscalculation**

# Example 1: Speel chek

A University of Pittsburgh study revealed that, for most students, computer spelling and grammar error checkers actually increased the number of errors they made.



- D. F. Galletta, A. Durcikova, A. Everard, and B. Jones. "Does Spell-Checking Software Need a Warning Label?" Communications of the ACM, pp. 82–86, July

# Example 2: High-Risk Inmates Mistakenly Released

In 2010, in the course of implementing a program meant to lessen prison overcrowding, more than 400 California prison inmates classified as “high risk of violence” were mistakenly released, owing to “computer errors.”

An additional 1,000 prisoners deemed to present a high risk of committing other crimes were also let out, stemming from those same errors.

None of the prisoners could be returned to prison or retroactively put on supervised parole, as they had already been granted “non-revocable parole.”





# Example 2: High-Risk Inmates Mistakenly Released

Under the law that created non-revocable parole, inmates are excluded if they are gang members, have committed sex crimes or violent felonies, or have been determined to pose a high risk to reoffend based on an assessment of their records behind bars.

The computer program that prison officials used to make that assessment did not (and could not) access an inmate's disciplinary history. The program also relied on a state-level Department of Justice system that recorded arrests but was missing conviction information for nearly half of the state's millions of arrest records.



- “Computer errors allow violent California prisoners to be released unsupervised” Los Angeles Times, May 26, 2011. Los Angeles Times, May 26, 2011.

# Example 3: Rent Miscalculation

Between September 2008 and May 2009, hundreds of low-income families living in public housing in New York City were charged too much rent because of a “computer error,” specifically an error in the program that calculated monthly bills.

For those nine months, the New York City Housing Authority ignored the complaints made by the renters that they were being overcharged.

Instead, it took to court and threatened with eviction many of those who failed to make the higher payments.



# Lessons Learned: Errors/Bugs enabling system malfunction

## Lesson 1 Avoid “automation bias” (Spell Check and Rent Miscalculation)

- *Automation bias* is a term often used to describe our documented habit of favoring or deferring to suggestions coming from automated decision-making systems and programs, to the point of ignoring contradictory information from sources not deploying automation, even when that information is correct.

# Lessons Learned: Errors/Bugs enabling system malfunction

**Lesson 2** *Ensure that the program has all the relevant data in the first place, not just that it operates correctly with any data it is already assumed to have (Inmates Released)*

- Sometimes failures of a program to query certain data can, in the broader context of the way the program is used, lead to flaws in the implementation of the program in its native context of use.
- Be sure to not just check the program in terms of how it operates with the data it has at the code-level, but also to consider whether the program itself is calling on all the relevant data it needs in order to be implemented correctly or appropriately in the broader context of use.

# 3. Errors/Bugs enabling system failure

**Example 1: Boeing 737 Max Crash**

**Example 2: Tesla 2016 Crash**

**Example 3: Patriot Missile System**

**Example 4: Ariane 5**

**Example 5: Robot Mission to Mars**

**Example 6: Flash Crashes**

# Example 1: Boeing 737 Max Crash



On October 29, 2018, a Boeing 737 Max airplane flying from Jakarta crashed into the Java Sea just 13 minutes after takeoff, taking the lives of all 189 passengers and crew. Subsequent investigations revealed flight control problems, failures of an “angle of attack” sensor, and other failures tied to a design flaw in a core system of the Max series. Less than 6 months later, in 2019, another 737 Max crashed only 6 minutes after takeoff, taking the lives of all 157 people aboard.

While the National Transportation Safety Committee’s (NTSC) final report lists nine contributing factors, the most serious seems to have been those pertaining to the “angle of attack” sensor and assumptions that were made about flight crew response to malfunctions.

Even though these assumptions were consistent with current industry guidelines, they turned out to be incorrect.

# Example 1: Boeing 737 Max Crash



As a flight expert explained:

*“In the two recent Boeing 737 Max crashes, it appears that the sensor sent wrong information to the flight control system indicating the angle of attack is too large and approaching stall, so the system steered the aircraft by pitching the nose down to correct the angle of attack. The pilots realized this was a wrong command and they tried to override the system, but it appears they could not figure out how to override it. So the planes went into a dive and eventually crashed... We are not at the phase where pilots should be left out of the cockpit. We think pilots still should fly the airplanes. Those two accidents are not due to poor design, it’s more **because of a judgment call in programming, a small error that made such a big impact.**”*

# Example 2: Tesla 2016 Crash

After 2014, Tesla started equipping its cars with “Autopilot” software that enabled the car to control its speed and steer, including an Automatic Emergency Braking (AEB) system.

They claimed the driver should use this autopilot “when the conditions are clear...with the expectation that the human driver will respond appropriately to a request to intervene...The driver is still responsible for, and ultimately in control of, the car” (Tesla).



*On May 7, 2016, Joshua Brown was killed when his Tesla crashed into the semi-trailer portion of a truck on a highway. The accident occurred as Brown’s Model S, with Autopilot engaged, was traveling east on a divided highway. The truck had been traveling in the opposite direction on the highway, and turned left in front of the Tesla. The semi-trailer portion was elevated enough off the ground that the car continued under it, shearing off its roof, after which the Tesla was sent veering off the road.*

- Quinn's Ethics for the Information Age, Chapter 8, for more extensive discussion of this incident.
- <https://electrek.co/2016/07/02/tesla-autopilot-mobileye-automatic-emergency-braking/>



# Example 2: Tesla 2016 Crash

After Brown's crash in 2016, the subsequent NTSB investigation determined that Autopilot was engaged for 37 minutes before the collision, during which time Brown's hands were on the steering wheel for only 25 seconds.

He received seven visual and audible warnings to put his hands back on the steering wheel.

As for the AEB system, when the truck turned in front of the Tesla and Brown failed to respond, ***why didn't it engage?***



- Quinn's Ethics for the Information Age, Chapter 8, for more extensive discussion of this incident.
- <https://electrek.co/2016/07/02/tesla-autopilot-mobileye-automatic-emergency-braking/>

# Example 2: Tesla 2016 Crash

Mobileye, which supplied the vision system for Autopilot, provided the explanation that the system was designed to avoid rear-end collisions, but not to avoid vehicles crossing laterally.

After Mobileye issued that statement, Tesla quickly released a “clarification” in which it noted its autopilot system relies on dozens of technologies to determine how it should respond to a particular situation.

According to Tesla, automatic braking did not engage because the trailer was white, making it difficult to see, and the trailer was tall, making the radar confuse it for an overhead sign.



- Quinn's Ethics for the Information Age, Chapter 8, for more extensive discussion of this incident.
- <https://electrek.co/2016/07/02/tesla-autopilot-mobileye-automatic-emergency-braking/>

# Lessons Learned: Errors/Bugs enabling system failure

**Lesson 1** *In the case of automated components of a system, even seemingly reasonable assumptions about the expected behavior of human operators – in the face of both malfunctions and ordinary operation – should be carefully scrutinized.*

**Lesson 2** *Smooth transition passing control between automated control and human operation should be carefully considered, both at the level of the code and system design.*

- If operators (pilots, drivers) do not place enough trust in an automated system, they may inject risk by intervening in system operation; but conversely, if pilots trust an automated system too much, they may either suffer from automation bias and fail to intervene when necessary, or lack sufficient time to act once they identify a problem.
- While clear transitions back to human control or manual override should be built into systems, experts have noted how in many emergency situations there simply may not be enough time for a human operator to retake control before an accident occurs. Thus, in general, **automated control systems should be designed in such a way that while humans always can take over in an emergency situation, humans are never needed to takeover in any of the emergency situations.**

**Lesson 3** *Sensor misinformation (as with the angle attack) or edge cases involving misidentification (as with the Tesla sensor mistaking the trailer portion of the truck for a sign) need to be considered in advance, and code-level mechanisms put in place to minimize the potential harm of such things in those rare circumstances where they do occur.*

# Example 3: Patriot missile system

The Patriot missile system was originally designed by the US Army to shoot down airplanes. In the Gulf War of 1991, the Army put the Patriot missile system to work defending against Scud missiles launched at Israel and Saudi Arabia.

At the end of the Gulf War, the Army claimed the Patriot missile defense system had been 95 percent effective at destroying incoming Scud missiles, but later analyses showed that perhaps as little as 9 percent of the Scuds were actually destroyed by Patriot missiles.

As it turns out, many Scuds simply fell apart as they approached their targets -- their destruction had nothing at all to do with the Patriot missiles launched at them.



# Example 3: Patriot missile system

The most significant failure of the Patriot missile system occurred during the night of February 25, 1991, when a Scud missile fired from Iraq hit a US Army barracks, killing 28 soldiers.

The Patriot missile battery defending the area never even fired at the incoming Scud. A report traced the failure of the Patriot system to a surprisingly trivial software error.

The Patriot system failed to track and intercept the incoming Scud due to a *truncation error* involving a loss of precision in converting integers to a floating point number representation!



# Example 3: Patriot missile system

The Patriot missile battery did detect the incoming Scud missile as it first came over the horizon. But in order to prevent the system from responding to false alarms, the computer had been programmed to check multiple times for the missile. It would do this by predicting the flight path of the incoming missile, directing the radar to focus on that area, and scanning a segment of that area, called a *range gate*, for the target. In this particular case, the program scanned the wrong range gate and thus failed to detect the missile.

**Why was the wrong range gate scanned by the program?** The prediction of where the Scud will next appear is a function of the Scud's velocity, of course determined by its change in position with respect to time, where time is updated in the Patriot's internal system clock in 100 ms intervals. Velocity was represented as a 24-bit floating point variable, and time as represented as a 24-bit integer, but both must be represented as 24-bit floating point numbers in order to predict where the Scud will next appear.

The conversion from integer time to real time results in a loss of precision that increases as the internal clock time increases. The error introduced by the conversion results in an error in the range gate computation, proportional to the target's velocity and the length of time that the system is running. The longer the system ran, the more these truncation errors added up – leading to disaster.

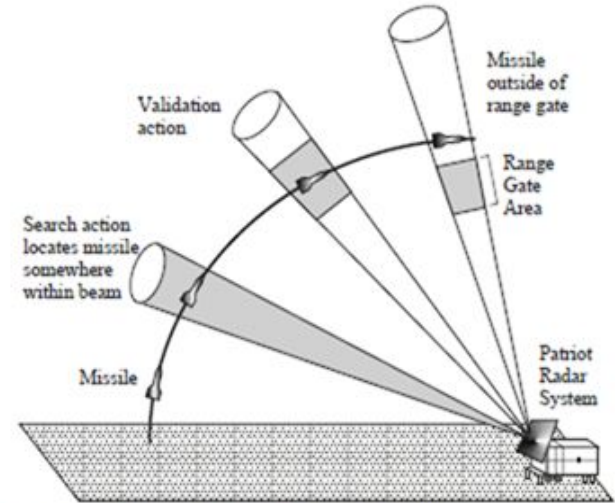


Figure 2-12 Effect of conversion error on range gate calculation.

# Example 4: Ariane 5

The Ariane 5 was a satellite launch vehicle designed by the French space agency.

During its maiden flight in 1996, about 40 seconds after takeoff, a software error caused the active and backup computers to shut down, resulting in the total loss of altitude control. The vehicle self-destructed.

While no one was injured, the uninsured rocket carried satellites worth \$500 million, making it the costliest error (in dollar terms) in history at the time.



# Example 4: Ariane 5



The software error that ultimately led to the crash was traced to a piece of code that converts a 64-bit floating-point value into a 16-bit signed integer. The input value that was to be converted ended up exceeding the range of what could be represented as a 16-bit signed integer. Moreover, there was no explicit exception handler to catch the overflow exception, so the uncaught exception crashed the entire software and the onboard computers. Arguably, this unhandled overflow exception was a main reason for the disaster.

As it turned out, this piece of code had been part of the 10-year-old software for the Ariane 4. Furthermore, the code that might have caught and handled the conversion errors had been disabled on the Ariane 4 – since, for that system, performance constraints made such overflow errors irrelevant – yet, as it turned out, those same performance constraints did not apply to the Ariane 5. The velocity reached by the Ariane 5 was much greater, and too high to be represented as a 16-bit integer, yet exception-handling had been suppressed.



# Example 4: Ariane 5



The Ada language (which was used for the Ariane's software) generates an exception if a floating-point number is too large to be converted to an integer; but the programmers had decided that this situation would never happen and thus didn't provide an exception handler.

The original programmer of the subprogram for converting a floating point number to a signed 16 bit number likely realized that the value of the floating point number to be converted must lie within a restricted range – but such restrictions on input values (preconditions for subprograms) were neither systematically derived nor well-documented.

A decision had also been made by the Ariane 5 team not to provide an overflow exception handler because the processor was already heavily loaded.

# Example 4: Ariane 5



The code was designed such that, in the event of any exception, the failure should lead the main processor to be shut down, and to switch to a backup. Since there was no exception handler in the code, when the overflow happened, the main software component failed and the system attempted to revert to the backup system. However, that backup system (based on the Ariane 4) was identical in hardware and software to the active one, and so it could not be activated because it had failed for exactly the same reason!

An irony was that the overflow conversion error that caused the issue occurred in a routine that had been re-used from the Ariane 4 vehicle, but was not even an appropriate computation to require for the Ariane 5. Moreover, as the computation that failed was not supposed to be required for Ariane 5, there was no associated requirement, so no tests were made of that portion of the software. The main code component in question had been validated for the Ariane 4, so no further testing with hardware and software, or test simulations with real data from the actual Ariane 5 trajectory, were ever performed.

- For more, see "Design by Contract: The Lessons of Ariane," <https://homepages.cwi.nl/~storm/teaching/reader/JezequelMeyer97.pdf>

# Lessons Learned: Errors/Bugs enabling system failure

## Lesson 1 *Apparently small errors can accumulate and have non-small consequences (Patriot Missile)*

- Take seriously the treatment of types, errors in final computations that can occur by, for instance, treating a floating point value like an integer, etc.

## Lesson 2 *It can be dangerous to reuse code, especially without a precise specification mechanism; any software element with a fundamental constraint should state this explicitly (and provide thorough documentation) -- without this, it is safer to redo rather than reuse (Ariane 5)*

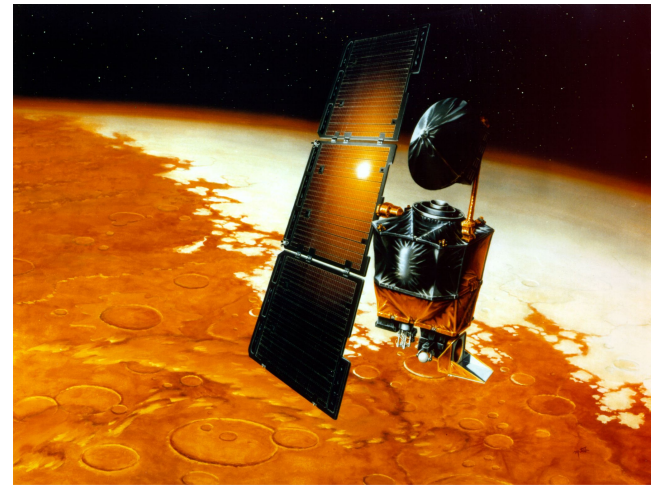
- As the Ariane 5 example demonstrates, assumptions that were valid when the code was originally written may no longer be true when the code is reused. Since some of these assumptions or constraints may not be stated explicitly or well-documented, the new team may not have the opportunity to check if these assumptions still hold true in the new system.
- In programming courses, one often learns to *reuse code*. The Ariane 5 incident provides a cautionary tale for what can go wrong if we aggressively reuse code in a thoughtless manner or do not provide proper documentation. Note that reusing code does not always increase the quality of the final product.

# Example 5: Robot Mission to Mars

NASA built the *Mars Climate Orbiter* to facilitate communications between Earth and automated probes on the surface of Mars, including the Mars Polar Lander. In 1999, the spacecraft was lost because of a miscommunication between two support teams on Earth.

The issue could be traced back to the fact that the flight operations team based in Colorado designed its software to use **English units**, so that its program output thrust in terms of *foot-pounds* units. The navigation team at the Jet Propulsion Laboratory in California, on the other hand, designed its software to use **metric units**, so its program expected thrust to be input in terms of *newtons*. One foot-pound equals 4.45 newtons.

On September 23, 1999, the Climate Orbiter neared Mars, and when it was time for the spacecraft to fire its engine to enter orbit, the Colorado team supplied thrust information to the California team, which then relayed it to the spacecraft. But there was a **units mismatch**, so the navigation team specified 4.45 times too much thrust, and the Orbiter flew too close to the surface of Mars and burned up in its atmosphere!



**Lessons Learned:** *Systems can fail on account of miscommunications among people.*

- In this case, the output of one program was incompatible with the input to the other program, and a poorly specified interface, together with miscommunication, allowed this error to remain undetected until after the spacecraft was destroyed.

# Example 6: Flash Crashes

- A **flash crash** is when there is a sudden and extreme plunge in prices of a stock, commodity, bond, currency, some other security, followed by a quick recovery. These are not generally crashes that occur due to any perceived change in the fundamental value of the stock.
- While there is some debate about the ultimate cause of flash crashes – which continue to be semi-regular occurrences in financial markets – it is evident that *algorithmic trading and the heavy use of automated trading computer programs and bots are one of the main causes.*



# 2010 Flash Crash

The **2010 Flash Crash** is the market crash that occurred on May 6, 2010. During the crash, leading US stock indices, including the Dow Jones, S&P 500, and Nasdaq Composite Index, tumbled and partially rebounded in less than an hour. The day was distinguished by extremely high volatility in trading. While the market indices managed to partially rebound in the same day, in the 15 minutes this whole debacle took to unfold the flash crash erased *almost \$1 trillion in market value!*

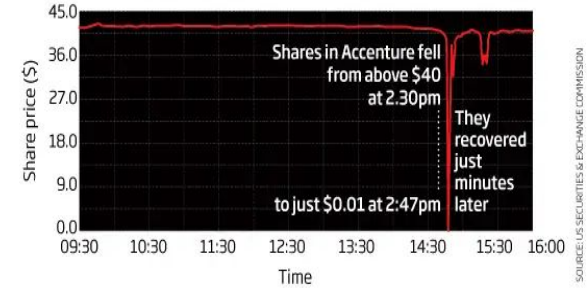
After the Flash Crash of 2010, there were congressional hearings and other investigations.

Since then, there have been at least 5 other very notable flash crashes, where the impact is felt market-wide. But to this day, across all markets, there are by some estimates at least 10 mini flash crashes *per day!*

Blink and you'll miss it

©NewScientist

On 6 May 2010 the US stock market went into freefall for a brief period. This "flash crash" is now blamed on high-frequency automated trading

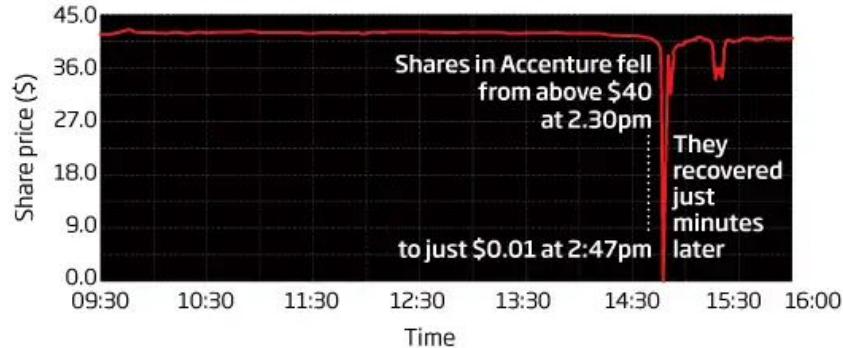


# 2010 Flash Crash

Blink and you'll miss it

©NewScientist

On 6 May 2010 the US stock market went into freefall for a brief period. This "flash crash" is now blamed on high-frequency automated trading



SOURCE: US SECURITIES & EXCHANGE COMMISSION

The massive decline in market prices was triggered by

- A single selling order of a large amount of S&P contracts,
- subsequent aggressive selling orders and automated strategies executed by high-frequency algorithms,
- small but not negligible delays suffered by the exchange computers in relaying accurate price data on to brokers and other trading platforms, together with
- some complex feedback loops involving trading bots and leading to accelerated downward price spiral.

# Breaking Down the Crash: Automatic Sell Orders

After the plunge in 2010, several reports indicated that while the event may have been initially triggered by an unusually large sell order for a particular big stock, this incited massive algorithmic trading automated orders to dump the stock via their “stop-loss” mechanism, which created a cascade of downward pressure.

A **stop-loss order** is essentially an automatic conditional trade order given to trigger a sale when a certain price level is reached to the downside (provided sufficient volume is present), and is typically used by traders to “cut losses.” Such orders are set to trigger once the price of the stock in question falls below the specified stop price threshold. Such orders are fundamentally designed to limit an investor’s loss on a position.

```
if ((price < stopLossThreshold) && (volume > minViableVol)):  
    MarketSellOrder;  
else;
```





# Breaking Down the Crash: Automatic Sell Orders

When a sale order for a stop-loss order is made, it can execute as either a market order or a limit order (though typically, it is as a market order).

A **market order** executes immediately, at whatever price the market is trading at that moment.

A **limit order** is a designated order “put on the order books,” that gets filled only if the further specified limit price is reached.

The **risk associated with stop-loss orders** can be particularly extreme if prices plunge as they do during a flash crash.



- A risk with a stop-loss order (executed as a market order) is that it triggers after a certain threshold is reached, but by the time the market order is executed, actually selling the stock, the price has slipped a great deal lower than that threshold, incurring bigger losses.
- No matter how quickly the price rebounds, once the stop-loss is triggered, it is triggered, and you sell at a loss (possibly extreme, if the order is executed at market price and the price moved a lot from the trigger level).

# Breaking Down the Crash: High-Frequency Traders

- High-frequency traders (HFTs), using more sophisticated algorithmic trading strategies, are another factor in flash crashes. HFTs make use of algorithms to carry out large transactions, executed at high speeds.
- Such HFTs have also used algorithms to carry out market manipulation activities like **‘spoofing’**, which is basically where algorithms place fake but large sell orders on the order books to intimidate or influence market participants, and then remove the order the moment before it gets filled. The purpose of spoofing can be to create the illusion of volume, artificially trick market participants into thinking that mass selling (or buying) is happening when it actually is not (since the orders are not meant to be filled), increase noise, clog exchanges, and generally outwit or intimidate competitors.



# Breaking Down the Crash: High-Frequency Traders

- There are further algorithmically-executed strategies, such as placing ‘**stub quotes**’ conditional on certain other factors (such as high volatility): a **stub quote** (also known as a **placeholder quote**) is an order to buy or sell shares that is deliberately set *far lower or higher* than the prevailing market price and are not generally expected to execute.
- In extreme situations – with extreme volatility (price swings) – stub quotes may actually get executed, and this will exacerbate any market volatility already present.



# Important Assumptions Made by the Bots

1. Traditionally, one would assume that volume (the number of shares of the stock that change hands during a specified time period) is a good indicator of liquidity (actual money present on that market), i.e., one assumes

## **Volume $\approx$ liquidity**

Especially before algorithmic trading entered the picture, this was a sensible assumption: if there's an increase in volume, it's because actual money is trading hands between people making a trade; if there is little to no volume, it shows that liquidity has dried up. But as algorithms and high-frequency traders became increasingly active in markets throughout the early 2000s, computer-driven trading programs would pass stocks back and forth in "hot potato" fashion, causing big increases in trading volume, without this often reflecting any actual increase in demand and the associated liquidity injections that would normally come with this.

2. Another normally reasonable assumption made by traders and their algorithms is that

## **the quoted price of an asset = its actual price**

In other words, it is assumed that price "oracles" feeding their algorithms the price don't lie.

# Observe Simplicity of Each Individual Algorithm/Assumption

- Each of the described algorithms or trading programs is remarkably simple, on its own.
- Looking closely at any one of them individually – say the *stub quote* order – there may be no reason to suppose that global system meltdown could arise from this.
- Moreover, the assumptions made in certain of the conditionals appear entirely sensible.
- However, *composed together*, and when (under unexpected circumstances) certain assumptions become erroneous, extreme and unanticipated effects can emerge on a large scale.

# When Sensible Assumptions Go Wrong

- The assumption that **trading volume is a good measure of liquidity** turned out to be **especially erroneous**, which had disastrous effects as the algorithms were composed together and acted in concert.

The increasingly aggressive selling and buying of large volumes of securities by the algorithmic trading bots resulted in enormous price volatility and high volume – which, mistaken for indicating high liquidity, led many algorithms to execute *market orders*, under the impression that there was liquidity/demand there to fill their orders. There was not, and so their orders would fill at prices that deviated from their stop-loss thresholds or expected price, often by ridiculous amounts.

# When Sensible Assumptions Go Wrong

- Furthermore, an analysis of trading on the exchanges during the moments immediately prior to the flash crash revealed **technical glitches in the reporting of prices** on the stock exchange that further contributed to the drying up of liquidity. According to this analysis, technical problems at the New York Stock Exchange (NYSE) led to delays as long as five minutes in NYSE quotes being reported on the system that reports the current quoted prices.
- At the same time, there were also errors in the prices of some stocks. Uncertain about prices, many market participants attempted to drop out of the market by posting stub quotes. Many high-frequency trading algorithms attempted to exit the market with their fast market orders, which began an “arms race” to the bottom – to the point that, *in a flash*, many orders got executed at stub quote levels, leading to ridiculously low prices in many stocks.

# Ultimate Cause of the 2010 Crash?

A peculiar combination of the algorithms compounded...

1. Initial **large (“fat finger”) sell order** →
2. Many stop thresholds of automated **stop-loss sell orders** get triggered →
3. **Increased trading volume** →
4. Since the automated trading programs operated under assumption that trading volume is a good measure of market liquidity (and liquidity generally means one can exit safely), this led to **increased sell pressure**, coupled with scarce liquidity triggering extreme price reduction



# Ultimate Cause of the 2010 Crash?

...and further exacerbating factors...

1. Since algorithmic trading bots were also **spoofing** the order books & there were **technical glitches in “price oracles”** →
2. The **market was actually illiquid**, contrary to what volume indicated →
3. As many high-frequency trading algorithms attempted to **exit the market with market orders as fast as possible** (which were **executed at the stub quote prices**) →
4. This had more of a **domino effect**

# Lessons Learned from Example 6

Individually, the programs that contributed to the flash crash were not particularly intelligent or sophisticated, e.g.,

*“sell if a certain price is reached and liquidity (= volume) is above some level.”*

**Lesson 1 Emergent Failures:** *Interactions between individually simple components can produce unexpected effects when composed.*

- **Systemic risk can build up** as new simple elements are introduced to a system, once a certain threshold of composability is met, extreme conditions arise, or under the presence of certain erroneous assumptions.
- In such cases, **the risks may not even be obvious until after something goes wrong.** In practice, such failures often pair with cascade effects – multiple failures that in isolation may have been minor or easily contained, but in concert produced aggregate disaster or consequences of great magnitude.



***Don't underestimate the power of composability and how many different protocols, code-components, and systems can interface and their vulnerabilities compound to create emergent failures.***

# Lessons Learned: Flash Crash

**Lesson 2** **The Perils of Consistency:** *Smart professionals might give instructions to a program based on an ordinarily (or previously) sensible and sound assumption – like, the assumption that trading volume is a reliable indicator of market liquidity – but this can produce catastrophic results when the program continues to act on the instruction (“volume = liquidity”) with a kind of strict logical consistency, even in unexpected situations where the assumption becomes invalidated.*

- The algorithm just does what it does. It doesn't care that we “didn't want” some unanticipated inappropriate action or that we didn't anticipate the new conditions that invalidated the assumption.
- We need to carefully scrutinize assumptions and even consider unlikely conditions that would invalidate seemingly sensible assumptions. Especially useful here are “red teams” and other groups of fresh eyes who look for exploitable aspects of the system.

# Lessons Learned: Flash Crash

**Lesson 3 Automate Safety:** *The need for pre-installed and automatically executing safety functionality – as opposed to reliance on runtime human supervision – can be crucial.*

- Some have put forth the theory that HFTs and automated mechanisms were actually a major factor in minimizing and reversing the flash crash in the end.
- In other words, while automation contributed to the incident, it also contributed to its swift resolution.
- The pre-programmed logic which ultimately suspended trading when prices moved too far was set to execute automatically because it had correctly anticipated that the triggering events could happen on a timescale too swift for humans to respond.

# Lessons Learned: Flash Crash

**Lesson 4** *Spread out risk!* If you're going to use, for instance, oracles feeding data (about price data) from one source, assume that if an agent can manipulate that data at the source, they will – and actually perform the calculations to figure out what will happen when they do! Then do what you can to *spread out risk*, by incorporating data feeds from multiple sources.

***Anticipate/Minimize Damage.*** If you cannot guarantee the elimination of certain risks, at least ensure that the total amount lost, or damage done to users or people affected by your code, is on a smaller scale or is contained.

*If dangers cannot be avoided, put caps on them!*

# Lessons Learned: Flash Crash

**Lesson 5** *Don't cut corners.* People often are most concerned with shipping as fast as possible, or getting things back up and running after a problem. But rushing these things can have bad consequences. *Prioritize safety over speed.*

- These days, there is a lot of pressure to ship quickly, even if this means making sacrifices. But the mantra “Break things first, ask questions later” is very questionable advice, ethically-speaking, especially when a lot is at stake.

# Meta-Lessons Learned from the Examples

- The goal is not so much avoiding all problems. After all, if you knew about them as problems to begin with, they would surely have been avoided in the first place!
- Rather, computerized systems should be designed in such a way that, when problems do arise, they can be resolved, and the system can be self-correcting.
- Moreover, as far as humans users and operators of that system are concerned, an important question to always keep in mind is:

*How do you write code, and align the incentives of the relevant users of such code, in such a way that, over the long-term, whenever problems or unintended results arise, the people involved have both the incentives and resources for repairing the system?*

# Assignment: Reflecting on Ariane 5

*See associated Assignment Sheet*