Steve's Research

Stephen Freund
Williams College

The Blue Screen of Death

- USS Yorktown
  - Smart Ship
    - 27 PCs
    - Windows NT 4.0
  - September 21, 1997:
    - data entry error caused a "Divide-By-0" error
    - entire system failed
    - ship dead in the water for over 2 hours

[Wired 1997]

USS Yorktown

- Ariane 5 Rocket
  - June 4, 1996
  - $800 million software failure
Mars Climate Orbiter

Purpose: Collect data, relay signals from Mars Polar Lander ($165M)
Failure: Smashed into Mars (1999)
Bug: Failed to convert English to metric units

Mars Polar Lander

Purpose: Lander to study the Mars climate ($120M)
Failure: Smashed into Mars (2000)
Bug: Spurious signals from sensors caused premature engine shutdown

North East Power Failure

Failure: Power grid failed across much of the North East. $6B losses (2001)
Bug: Timing bug in Ohio power plant

Online Trading Software

Purpose: automatic high-frequency trading
Failure: DOW drops 9.2%, equity markets collapse (2010)
Bug: Bad modeling, and no fail-stops to prevent flooding market with sell orders

7

Patriot Missile

Purpose: Intercept incoming missiles
Failure: missed SCUD missile that killed 28 US soldiers (1991)
Bug: Incorrect calculation of distance to target

8

Therac25 Radiation Therapy

Purpose: Computer-controlled radiation therapy machine
Failure: gave fatal radiation doses to 2 cancer patients (1986)
Bug: race condition (timing bug)

Heartbleed SSL Attack

Purpose: OpenSSL is widely used cryptographic library
Failure: Library could leak sensitive information, including keys (2014)
Bug: Buffer overrun

9

Teslas

Purpose: Self-Driving Cars
Failure: Fatal Crash (2016)
Bug: Failed to distinguish a white tractor-trailer crossing the highway against a bright sky. (Other fatal accidents have followed...)

10

Teslas

Purpose: Self-Driving Cars
Failure: Fatal Crash (2016)
Bug: Failed to distinguish a white tractor-trailer crossing the highway against a bright sky. (Other fatal accidents have followed...)

Buffer Overruns

[2]: def f(array, index):
   array[index] = 42
[3]: elems = make_array(1,2,3,4,5,6,7,8,9,10)
x = 100
print(x)
f(elems,6)
print(x)
100
12
Buffer Overruns

```c
void f(int array[], int index) {
    array[index] = 42;
}

int main() {
    int x = 100;
    int elems[10] = { 1,2,3,4,5,6,7,8,9,10};
    printf("%d\n", x);
    f(elems, 6);
    printf("%d\n", x);
    f(elems, 11);
    printf("%d\n", x);
}
```

$gcc array.c$

$.

```
100
100
42
```

[https://www.informationisbeautiful.net/visualizations/million-lines-of-code/](https://www.informationisbeautiful.net/visualizations/million-lines-of-code/)
Managing Software Complexity

People

Process

Tools

Research on Program Checkers

Identify Type of Bug
- Bad unit conversion
- Buffer overrun
- Data Race
- ...

Design Checking Tool
- static or dynamic?
- precision?
- scalability?
- performance?
- usability?

Validate Technique
- check real software
- find bugs...

Source Code (Static) Checkers

Source Code

Good

Bad

Good Program
Has No Buffer Overruns
Bad Program
Has Buffer Overrun

No algorithm can precisely compute if a program is “Good” or “Bad”
- Undecidability of the Halting Problem [Turing 1936]

Source Code (Static) Checkers

Source Code

Verifiable

Not Verifiable

Good Program
Has No Buffer Overruns
Bad Program
Has Buffer Overrun
Verifiable Program
Can Prove No Buffer Overruns

+ Catch many errors prior to testing
- Must reject some good programs...

Dynamic Checkers

Application

Good

Bad

+ Can discern Good vs. Bad precisely, but...
- only during the tests performed
- Performance

New Languages, Programming Models

Identify Type of Bug
- Bad unit conversion
- Buffer overrun
- Data Race
- ...

Design Checking Tool
- static or dynamic?
- precision?
- scalability?
- performance?
- usability?

Validate Technique
- check real software
- find bugs...
Multithreading and Multicore CPUs

Concurrent Programming With Threads

Concurrent Programming With Threads

Concurrent Programming With Threads

Concurrent Programming With Threads

y_hat = a * table.column('x') + b

Divide array into four pieces and do multiplications and additions for each piece on a different thread.

Bank Server

Amazon.com

Thread 1

Thread 2

Thread 3

Thread 4
Thread Interference

• Race Conditions
  two concurrent unsynchronized accesses, at least one write

Thread A
  ... t1 = bal;
  bal = t1 + 10;
  ...

Thread B
  ... t2 = bal;
  bal = t2 - 10;
  ...

Thread Interference

• Race Conditions
  two concurrent unsynchronized accesses, at least one write

Thread A
  t1 = bal;
  bal = t1 + 10;
  ...

Thread B
  t2 = bal;
  bal = t2 - 10;
  ...

Controlling Thread Interference:
Mutual Exclusion Locks

Thread A
  acq(m);
  t1 = bal;
  bal = t1 + 10;
  rel(m);

Thread B
  acq(m);
  t2 = bal;
  bal = t2 - 10;
  rel(m);

Controlling Thread Interference:
Mutual Exclusion Locks

Thread A
  acq(m);
  t1 = bal;
  bal = t1 + 10;
  rel(m);

Thread B
  acq(m);
  t2 = bal;
  bal = t2 - 10;
  rel(m);
In Real Life...

- Huge systems
- Many threads
- Many mutual exclusion locks
- Many other types of synchronization
- Programmers make mistakes

• Many data-race bugs

Examples

```java
void deposit(int n) {  
synchronized(m) {  
    t1 = bal;  
    bal = t1 + n;  
  }
}
```

```java
void deposit(int n) {  
synchronized(m) {  
    t1 = bal;  
    bal = t1 + n;  
  }
}
```

FastTrack '10, RedCard '13, SlimState '15, BigFoot '17

```java
int x;

def f() {
  synchronized ..
}
```

RcJava '02

```java
int x guarded_by m1;  
requires m1, m2  
def f() {  
  synchronized ..
}
```

Data Race on var x

Data Race on var x
In Real Life...

- Huge systems
- Many threads
- Many mutual exclusion locks
- Many other types of synchronization
- Many subtle requirements for atomicity
- Programmers make mistakes

- Many atomicity bugs
Program Synthesis

- How to generate candidate versions?
- How to verify candidates are correct?
- How to pick most performant?

Programming Languages And Analysis Tools

- language design
- theoretical foundations
- proving theorems
- systems development
- performance modeling
- experimental validation