Program Verification and Ethics of Performance Tuning

Aarti Gupta

Acknowledgements:

Andrew Appel, Ethics of Extreme Performance Tuning

Agenda

Famous bugs

Common bugs

Testing (from lecture 6)

Reasoning about programs

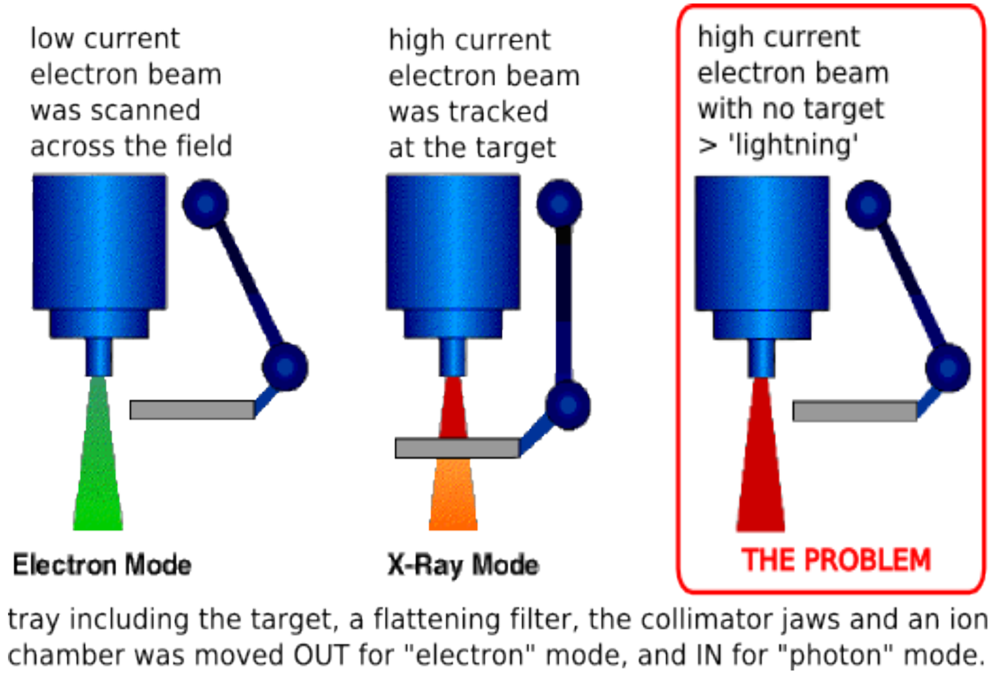
Ethics of performance tuning

Famous Bugs

The first bug: A moth in a relay (1945)
At the Smithsonian (currently not on display)

(in)Famous Bugs

- Safety-critical systems

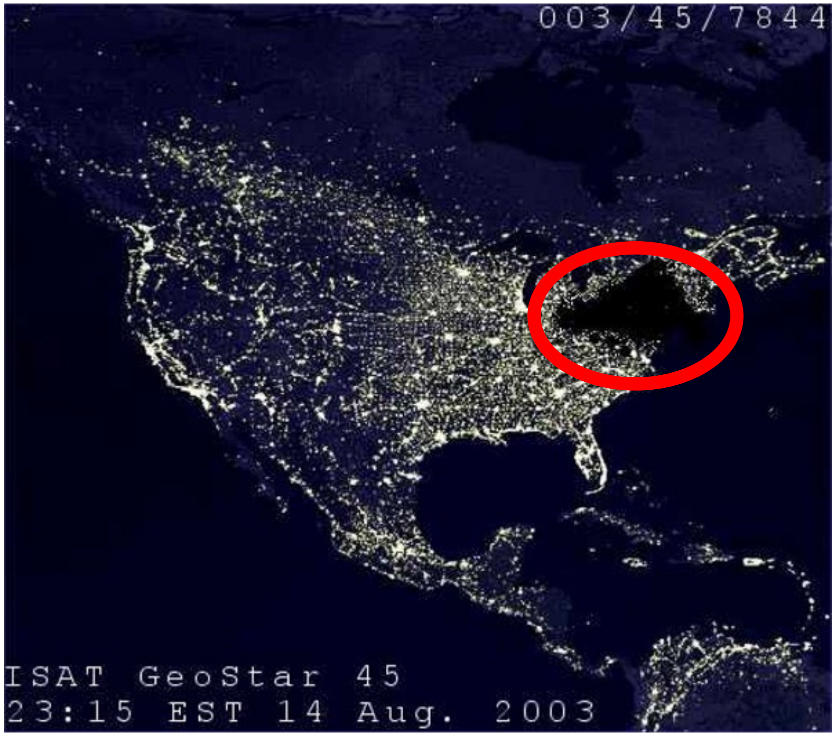


Therac-25 medical radiation device (1985)
At least 5 deaths attributed to a race condition in software

(in)famous bugs

- mission-critical systems

Ariane-5 self-destruction (1995)
***SW interface issue*, backup failed**
cost: $400M payload

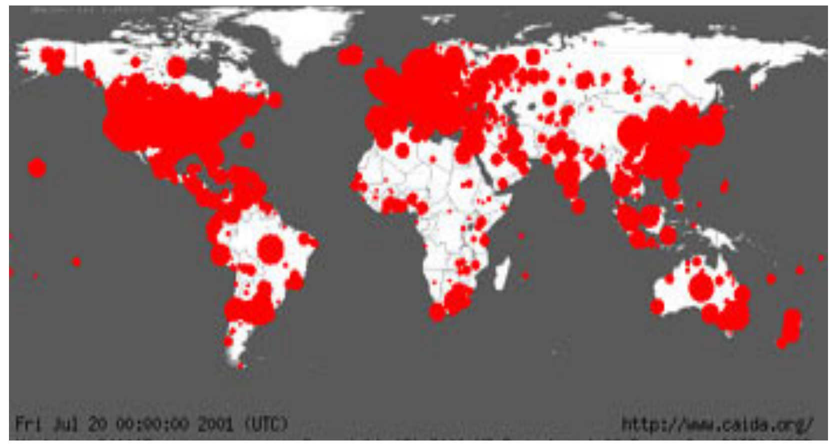


the Northeast Blackout (2003)
***race condition* in power control software**
cost: $4B

(in)famous bugs

- commodity hardware / software

Pentium bug (1994)
float computation errors
cost: $475M



Code Red worm on MS IIS server (2001)
***buffer overflow* exploited by worm**
Infected 359k servers
cost: >$2B

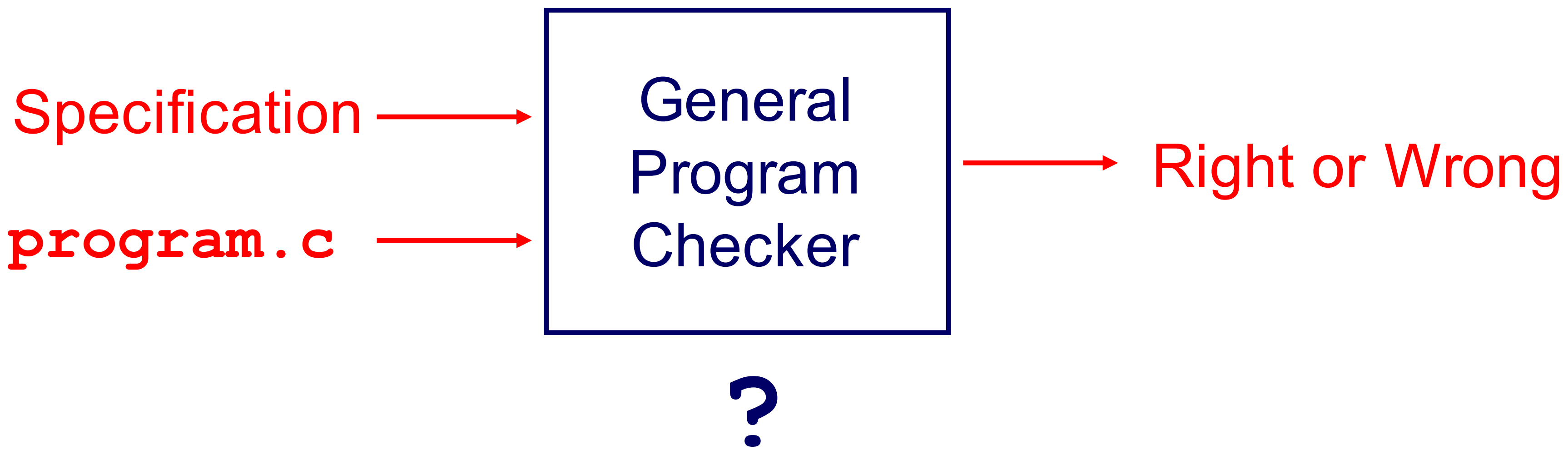
heartbleed **(2014)**

Common Bugs

- Runtime bugs
  - Null pointer dereference (access via a pointer that is Null)
  - Array buffer overflow (out of bound index)
    - Can lead to security vulnerabilities
  - Uninitialized variable
  - Division by 0
- Concurrency bugs
  - Race condition (flaw in accessing a shared resource)
  - Deadlock (no process can make progress)
- Functional correctness bugs
  - Input-output relationships
  - Interface properties
  - Data structure invariants
  - …

Program Verification

Ideally: Prove that any given program is correct



In general: Undecidable

This lecture: For some (kinds of) properties, a Program Verifier can provide a proof (if right) or a counterexample (if wrong)

Program Testing (Lecture 6)

Pragmatically: Convince yourself that a **specific** program **probably** works

"Program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence."
– Edsger Dijkstra

Path Testing Example (Lecture 6)

Example pseudocode:

```
if (condition1)
    statement1;
else
    statement2;
…
if (condition2)
    statement3;
else
    statement4;
…
```

Path testing:

Should make sure all logical paths are executed

How many passes through code are required?

Four paths for four combinations of (condition1, condition 2): TT, TF, FT, FF

- Simple programs => maybe reasonable
- Complex program => combinatorial explosion!!!
  - Path test code fragments

Agenda

Famous bugs

Common bugs

Testing (from lecture 6)

Reasoning about programs

Ethics of performance tuning

Reasoning about Programs

```
1 int factorial(int x) {
2   int y = 1;
3   int z = 0;
4   while (z != x) {
5     z = z + 1;
6     y = y * z;
7   }
8   return y;
9 }
```

Example:
factorial program

Check:
If x >= 0, then y = fac(x)
(fac is the mathematical function)

- Try out the program, say for x=3
  - At line 4, before executing the loop: x=3, y=1, z=0
  - Since z != x, we will execute the while loop
  - At line 4, after 1st iteration of loop: x=3, z=1, y=1
  - At line 4, after 2nd iteration of loop: x=3, z=2, y=2
  - At line 4, after 3rd iteration of loop: x=3, z=3, y=6
  - Since z == x, exit loop, return 6: It works!

Reasoning about Programs

```
1 int factorial(int x) {
2   int y = 1;
3   int z = 0;
4   while (z != x) {
5     z = z + 1;
6     y = y * z;
7   }
8   return y;
9 }
```

Example:
factorial program

Check:
If x >= 0, then y = fac(x)

- Try out the program, say for x=4
  - At line 4, before executing the loop: x=4, y=1, z=0
  - Since z != x, we will execute the while loop
  - At line 4, after 1st iteration of loop: x=4, z=1, y=1
  - At line 4, after 2nd iteration of loop: x=4, z=2, y=2
  - At line 4, after 3rd iteration of loop: x=4, z=3, y=6
  - At line 4, after 4th iteration of loop: x=4, z=4, y=24
  - Since z == x, exit loop, return 24: It works!

Reasoning about Programs

```
1 int factorial(int x) {
2   int y = 1;
3   int z = 0;
4   while (z != x) {
5     z = z + 1;
6     y = y * z;
7   }
8   return y;
9 }
```

Example:
factorial program

Check:
If x >= 0, then y = fac(x)

- Try out the program, say for x=1000
  - At line 4, before executing the loop: x=1000, y=1, z=0
  - Since z != x, we will execute the while loop
  - At line 4, after 1st iteration of loop: x=1000, z=1, y=1
  - At line 4, after 2nd iteration of loop: x=1000, z=2, y=2
  - At line 4, after 3rd iteration of loop: x=1000, z=3, y=6
  - At line 4, after 4th iteration of loop: x=1000, z=4, y=24

Want to keep going on???

Lets try some mathematics …

```
1 int factorial(int x) {
2   int y = 1;
3   int z = 0;
4   while (z != x) {
5     z = z + 1;
6     y = y * z;
7   }
8   return y;
9 }
```

Example:
factorial program

Check:
If x >= 0, then y = fac(x)

- Annotate the program with assertions [Floyd 67]
  - Assertions (at program lines) are expressed as (logic) formulas
    - Here, we will use standard arithmetic
  - Meaning: Assertion is true before that line is executed
    - E.g., at line 3, assertion y=1 is true
- For loops, we will use an assertion called a loop invariant
  - Invariant means that the assertion is true in each iteration of loop

Loop Invariant

```
1 int factorial(int x) {
2   int y = 1;
3   int z = 0;
4   while (z != x) {
5     z = z + 1;
6     y = y * z;
7   }
8   return y;
9 }
```

Example:
factorial program

Check:
If x >= 0, then y = fac(x)

- Loop invariant (assertion at line 4): y = fac(z)
- Try to *prove by induction* that the loop invariant holds
- Use induction over n, the number of loop iterations

Aside: Mathematical Induction

Example:

- Prove that sum of first n natural numbers = n * (n+1) / 2

Solution: Proof by induction

- Base case: *Prove* the claim for n=1
  - LHS = 1, RHS = 1 * 2 / 2 = 1, claim is true for n=1
- Inductive hypothesis: *Assume* that claim is true for n=k
  - i.e., 1 + 2 + 3 + … k = k * (k+1) / 2
- Induction step: Now *prove* that the claim is true for n=k+1
  - i.e., 1 + 2 + 3 + … k + (k+1) = (k+1) * (k+2) / 2

  LHS = 1 + 2 + 3 + ... k + (k+1)
  = (k * (k+1))/2 + (k+1) … by using the inductive hypothesis
  = (k * (k+1))/2 + 2*(k+1)/2
  = ((k+2) * (k+1)) / 2
  = RHS
- Therefore, claim is true for all n

Loop Invariant

```
1 int factorial(int x) {
2   int y = 1;
3   int z = 0;
4   while (z != x) {
5     z = z + 1;
6     y = y * z;
7   }
8   return y;
9 }
```

Example:
factorial program

Check:
If $x >= 0$, then $y = fac(x)$

- Loop invariant (assertion at line 4): $y = fac(z)$
- Try to *prove by induction* that the loop invariant holds
  - Base case: First time at line 4, $z=0$, $y=1$, $fac(0)=1$, $y=fac(z)$ holds √
  - Induction hypothesis: *Assume* that $y = fac(z)$ at line 4
  - Induction step: In next iteration of the loop (when $z!=x$)
    - $z' = z+1$ and $y'= fac(z)*z+1 = fac(z')$ (z'/y' denote updated values)
    - Therefore, at line 4, $y'=fac(z')$, i.e., loop invariant holds again √

Proof of Correctness

```
1 int factorial(int x) {
2   int y = 1;
3   int z = 0;
4   while (z != x) {
5     z = z + 1;
6     y = y * z;
7   }
8   return y;
9 }
```

Example:
factorial program

Check:
If x >= 0, then y = fac(x)

- We have proved the loop invariant (assertion at line 4): y = fac(z) √
- What should we do now?
  - Case analysis on loop condition
  - If loop condition is true, i.e., if (z!=x), execute loop again, y=fac(z)
  - If loop condition is false, i.e., if (z==x), exit the loop
    - At line 8, we have y=fac(z) AND z==x, i.e., y=fac(x)
    - Thus, at return, y = fac(x)
- Proof of correctness of the factorial program is now done √

Program Verification

- Rich history in computer science
- *Assigning Meaning to Programs* [Floyd, 1967]
  - Program is annotated with assertions (formulas in logic)
  - Program is proved correct by reasoning about assertions

Turing Award 1978

- *An Axiomatic Basis for Computer Programming* [Hoare, 1969]
  - Hoare Triple: {P} S {Q}
  - Meaning: If S executes from a state where P is true, and if S terminates, then Q is true in the resulting state
  - For our example: {x >= 0} y = factorial(x); {y = fac(x)}

Turing Award 1980

Program Verification

- Proof Systems
  - Perform reasoning using logic formulas and rules of inference
- Hoare Logic [Hoare 69]
  - Inference rules for assignments, conditionals, loops, sequence
  - Given a program annotated with preconditions, postconditions, and loop invariants
    - Generate Verification Conditions (VCs) automatically
    - If each VC is “valid”, then program is correct
    - Validity of VC can be checked by a theorem-prover
- Question: Can these preconditions/postconditions/loop invariants be generated automatically?

Automatic Program Verification

- Question: Can these preconditions/postconditions/loop invariants be generated automatically?
- Answer: Yes! (in many cases)
- Techniques for deriving the assertions automatically
  - Model checkers: based on exploring "states" of programs
  - Static analyzers: based on program analysis using "abstractions" of programs
  - … many other techniques
- Still an active area of research (after more than 45 years)!

Model Checking

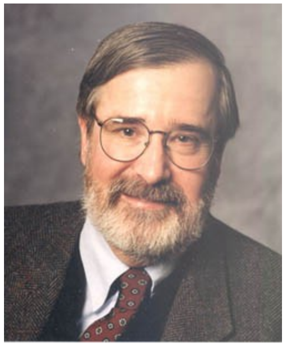
- Temporal logic
  - Used for specifying correctness properties
  - [Pnueli, 1977]

Turing Award 1996

- Model checking

Turing Award 2007

  - Verifying temporal logic properties by state space exploration
  - [Clarke & Emerson, 1981] and [Queille & Sifakis, 1981]



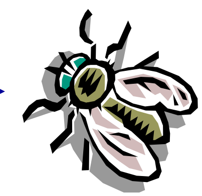
F-Soft Model Checker

Automatic tool for finding bugs in large C/C++ programs (NEC)

```
1: void pivot_sort(int A[], int n){
2: int pivot=A[0], low=0, high=n;
3: while ( low < high ) {
4:      do {
5:          low++ ;
6:      } while ( A[low] <= pivot ) ;
7:      do {
8:          high - - ;
9:      } while ( A[high] >= pivot );
10:     swap(&A[low],&A[high]);
11:   }
12: }
```

Array Buffer Overflow?

F-Soft



counterexample trace

Line 1: n=2, A[0]=10, A[1]=10

Line 2: pivot=10, low=0, high=2

Line 3: low < high ? YES

Line 5: low = 1

Line 6: A[low] <= pivot ? YES

Line 5: low = 2

Line 6: A[low] <= pivot ?

Buffer Overflow!!!

Summary

- Program verification
  - Provide *proofs of correctness* for programs
  - Testing *cannot* provide proofs of correctness (unless exhaustive)
- Proof systems based on logic
  - Users annotate the program with assertions (formulas in logic)
  - Theorem-provers: user-guided proofs of correctness
  - Automatic verification: automate the search

Active area of research!

COS 516 in Fall ’17: Automatic Reasoning about Software

COS 510 in Spring ’18: Programming Languages

Cat-and-mouse regarding the buffer overrun problem

1972

Niklaus Wirth designs Pascal language,

with supposedly ironclad array-bounds checking.

Turing award 1984

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 7, 685–696 (1977)

Ambiguities and Insecurities in Pascal

J. WELSH, W. J. SNEERINGER* AND C. A. R. HOARE†
Department of Computer Science, Queen's University, Belfast BT7 1NN, N. Ireland

Turing award 1980

1978

Robin Milner designs ML programming language, with provably secure type-checking.

Turing award 1991

1988

Everything is still written in C . . .

Robert T. Morris, graduate student at Cornell, exploits **buffer overruns** in Internet hosts (sendmail, finger, rsh) to bring down the entire Internet.

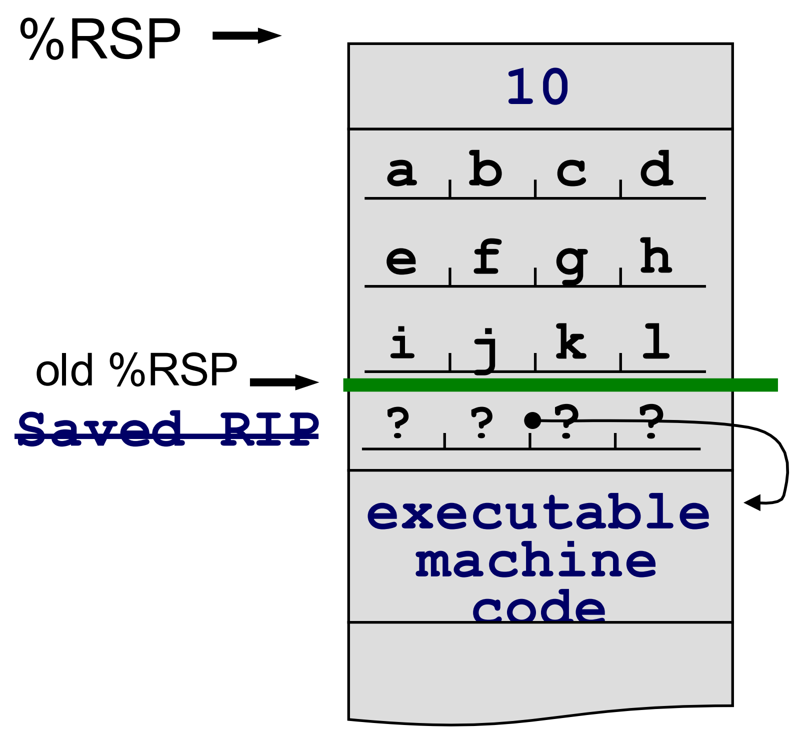
. . . became the first person convicted under the then-new Computer Fraud and Abuse Act.

(400 hours community service. Now an MIT prof.)

Cleverly malicious? Maliciously clever? Buffer overrun

```
% a.out
What is your name?
abcdefghijkl????executable-machine-code...
How may I serve you, master?
%
```

```c
#include <stdio.h>
int main(int argc, char **argv) {
   char name[12];  int i;
   printf("What is your name?\n");
   for (i=0; ; i++) {
      int c = getchar();
      if (c=='\n' || c ==EOF) break;
      name[i] = c;
   }
   name[i]='\0';
   printf("Thank you, %s.\n", name);
   return 0;
}
```



1990s

Everything is *still* written in C . . .

Buffer overrun attacks proliferate like crazy

“Solution:”

Every time the OS “execvp”s a new process,

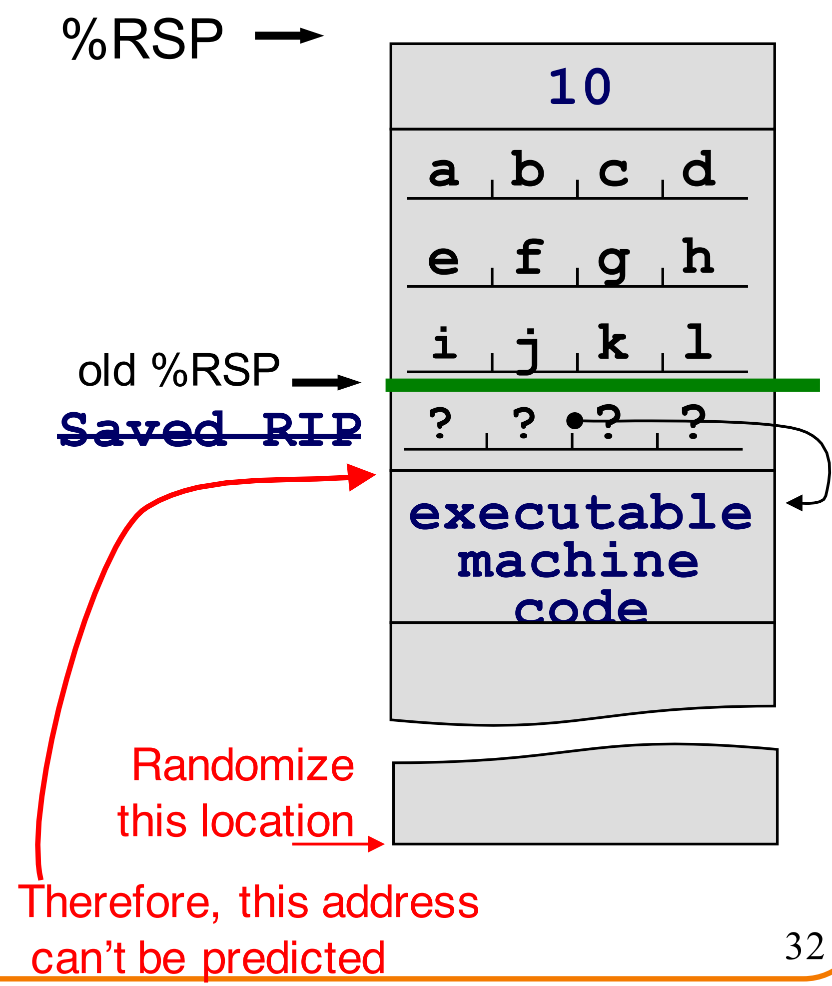
randomize the address of the base of the stack.

That way, code-injection attacks can’t predict what address to jump to!

Buffer overrun with random stack-start

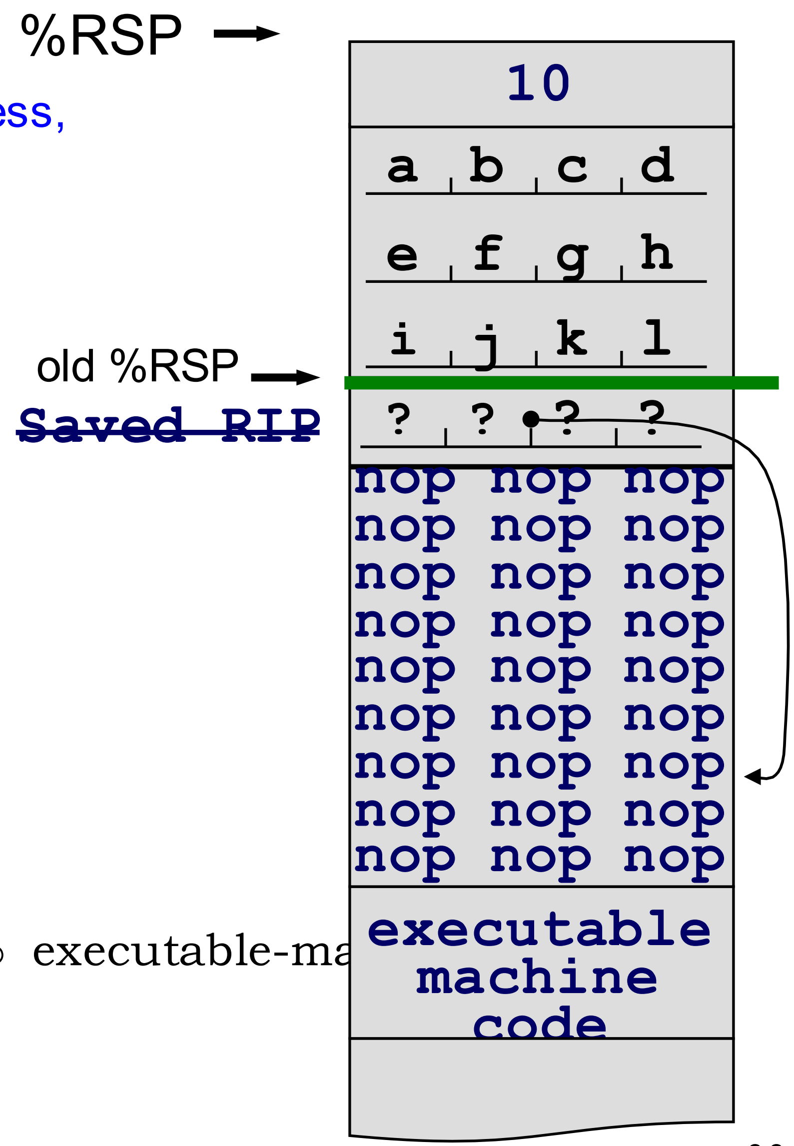
```
% a.out
What is your name?
abcdefghijkl????executable-machine-code...
How may I serve you, master?
%
```

```
#include <stdio.h>
int main(int argc, char **argv) {
  char name[12];  int i;
  printf("What is your name?\n");
  for (i=0; ; i++) {
    int c = getchar();
    if (c=='\n' || c ==EOF) break;
    name[i] = c;
  }
  name[i]='\0';
  printf("Thank you, %s.\n", name);
  return 0;
}
```



The nop-sled attack

“Solution:” Every time the OS “execvp”s a new process, randomize the address of the base of the stack. That way, code-injection attacks can’t predict what address to jump to!




```
% a.out
What is your name?
abcdefghijkl????nop nop nop nop nop nop executable-ma
How may I serve you, master?
%
```

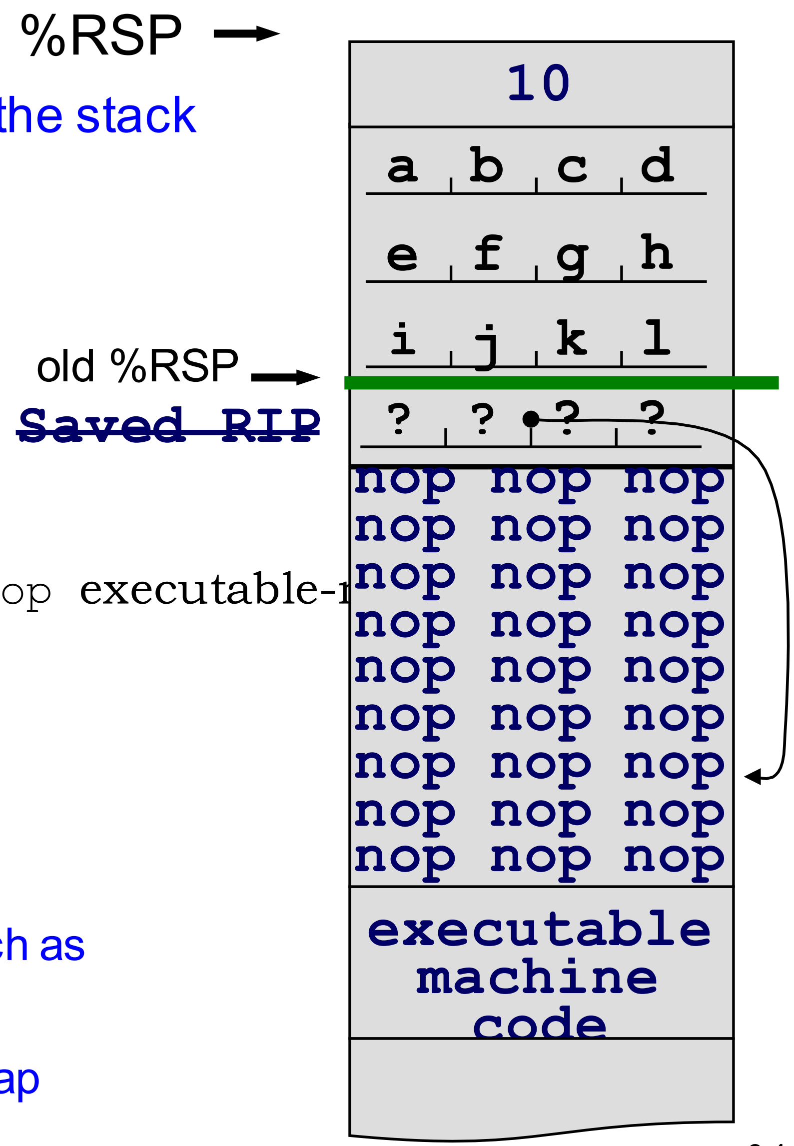
“Solution:” hardware permissions

“Solution:” In the virtual memory system, mark the stack region “no-execute”

(required inventing new hardware mechanism!)

```
% a.out
What is your name?
abcdefghijkl????nop nop nop nop nop nop executable-r
```

Segmentation violation



BUT:

(1) doesn’t protect against return-to-libc attacks (such as the “B” version of homework 5

(2) doesn’t protect against code injection into the heap (such as the “A” version of homework 5)

“Solution:” more hardware permissions

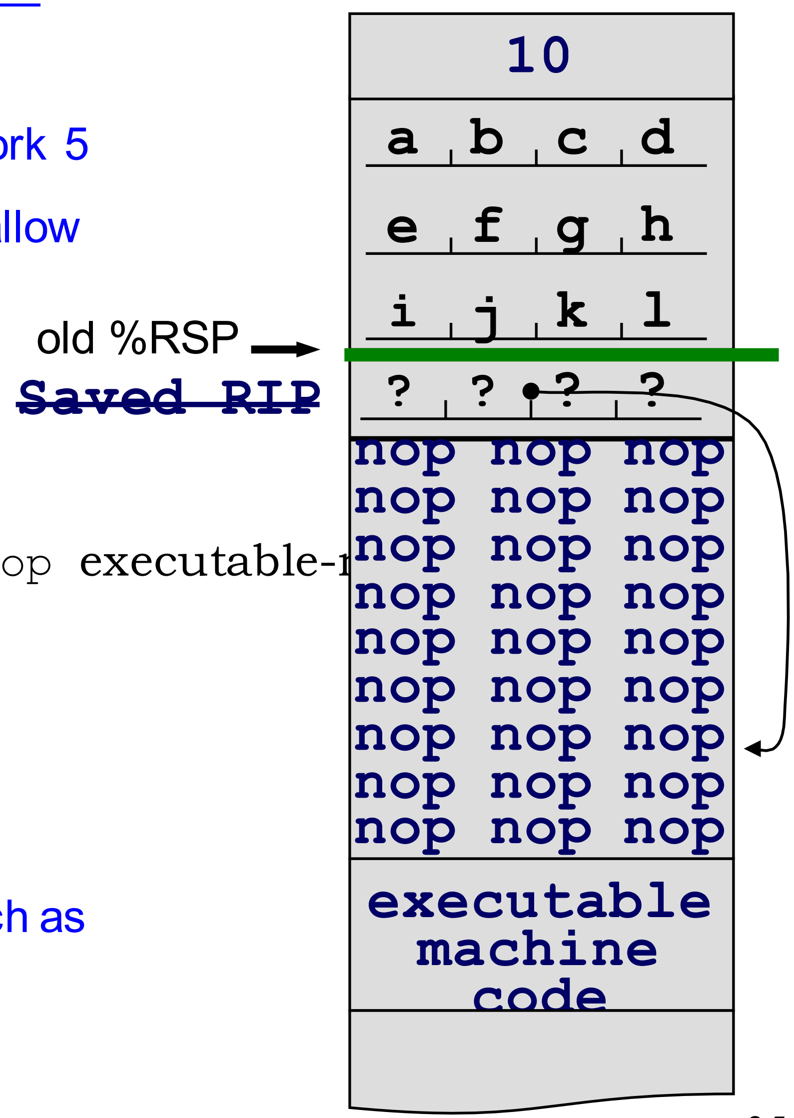
“Solution:” In the virtual memory system, mark the **BSS** region “no-execute.”

This DOES protect against the “A” version of homework 5

(and we had to specifically disable this protection to allow you to have your fun)

```
% a.out
What is your name?
abcdefghijkl????nop nop nop nop nop nop executable-r
```

Segmentation violation



BUT:

(1) doesn’t protect against return-to-libc attacks (such as the “B” version of homework 5

"Solution:" canary values

"Solution:" Check whether the canary has been overwritten,

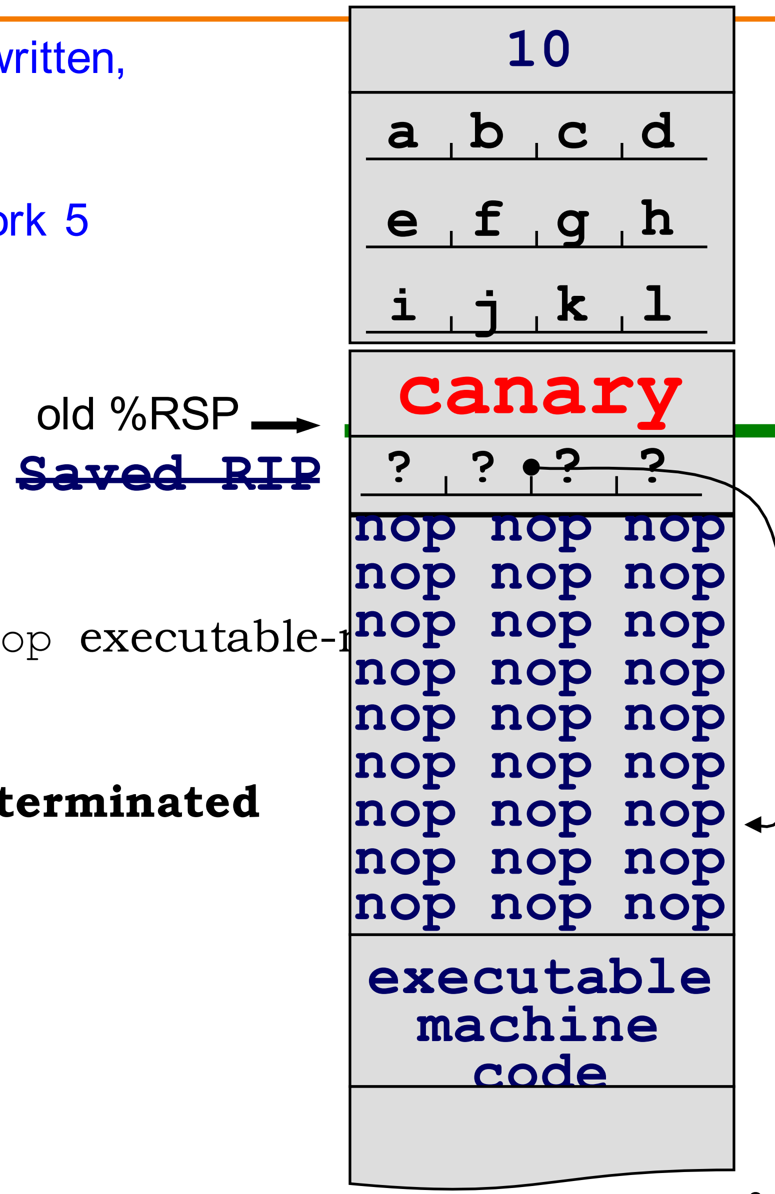
just before returning from the function.

This DOES protect against the "A" version of homework 5

This DOES protect against return-to-libc attacks

```
% a.out
What is your name?
abcdefghijkl????nop nop nop nop nop nop executable-r
```

Stackguard detected an attack, execution terminated



BUT:

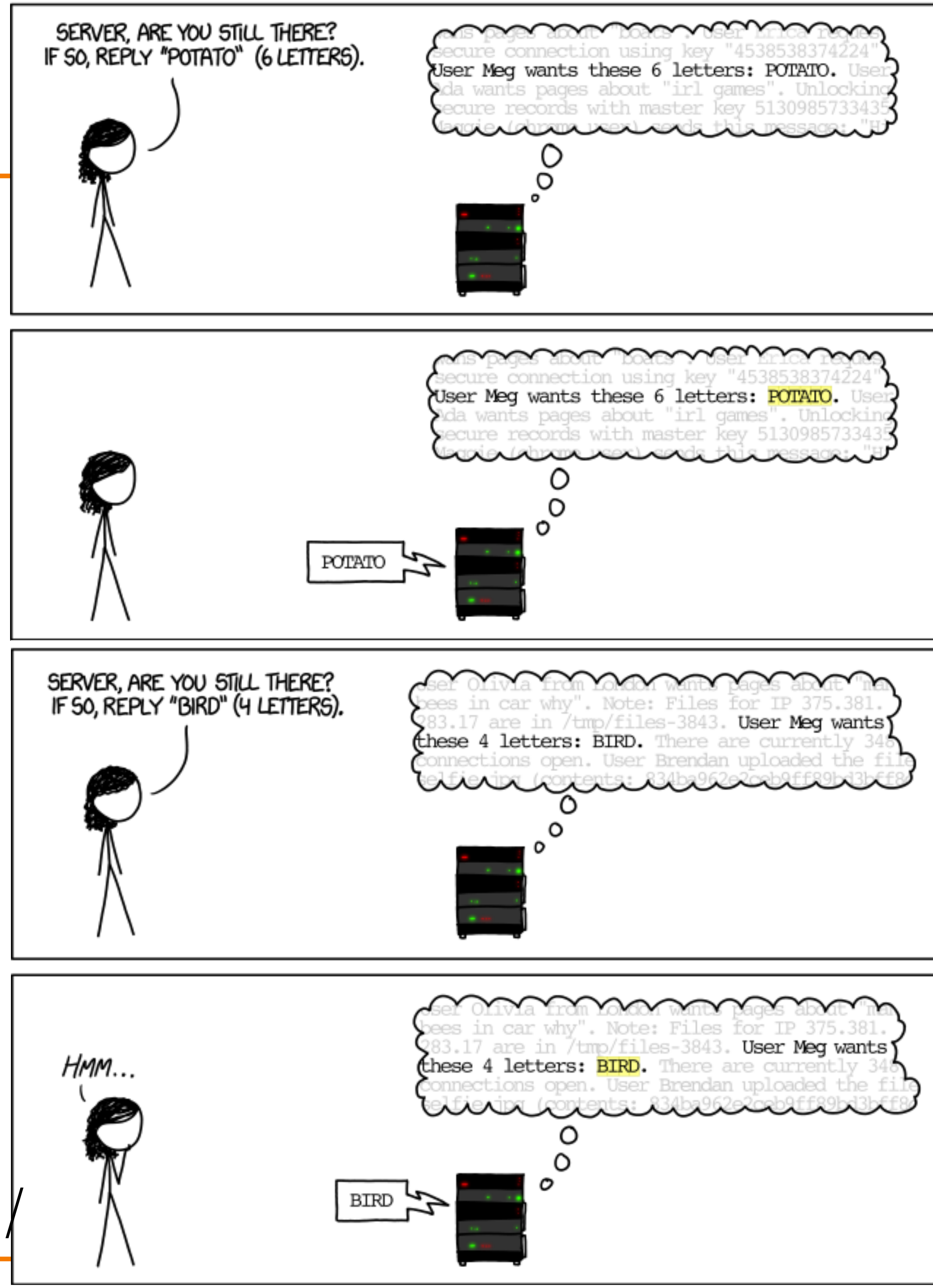
(1) There are still ways to defeat it

(2) Costs overhead, never much caught on

Heartbeat

Component of OpenSSL

Used across the Internet

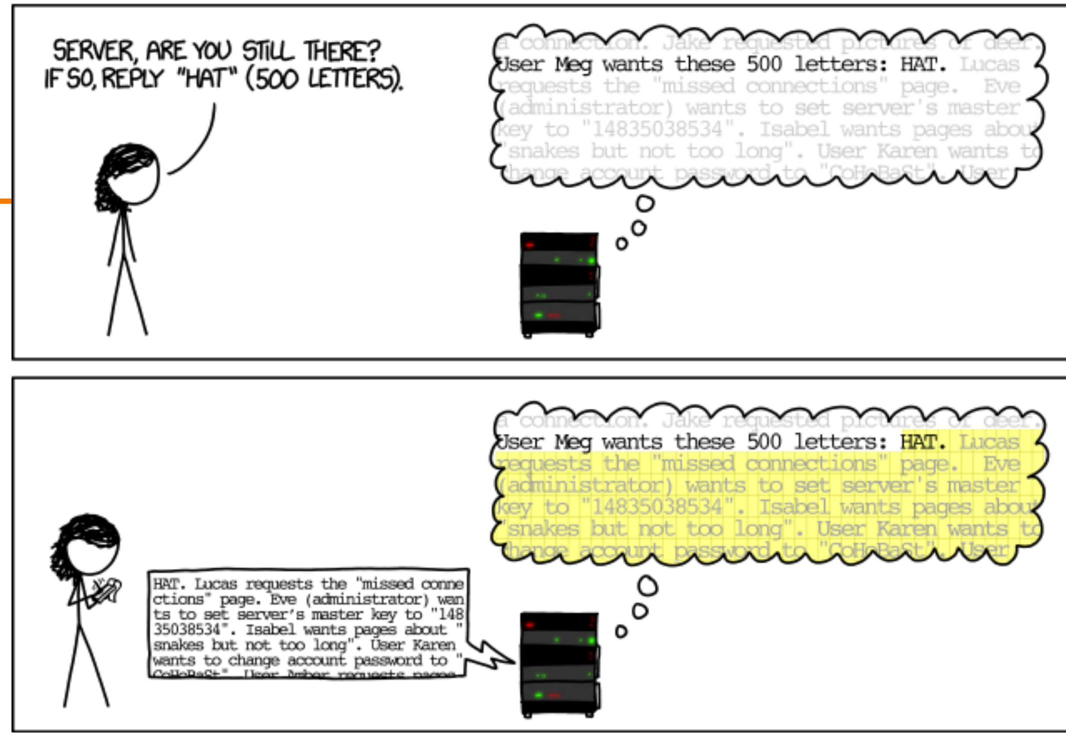


http://xkcd.com/1354/

Bug in OpenSSL

If strlen() doesn't match given length . . .

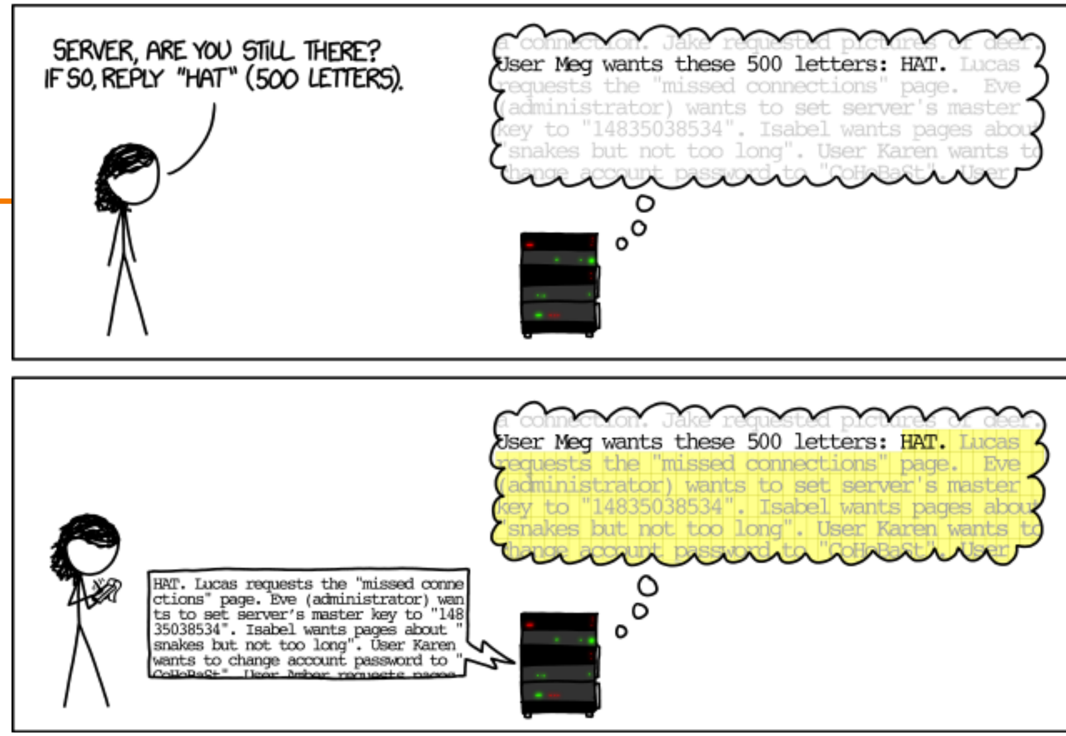
buffer overrun



HeartBleed

First Internet bug report with:

- catchy name,
- logo
- web site



Consequence:
Read up to 64 kilobytes from your OS address space, send it to attacker.

If those happen to contain crypto keys or other secret info, you're hacked!

http://xkcd.com/1354/

Those protections don't work against HeartBleed

Stack randomization: doesn't protect.
Stack no-execute: doesn't protect
BSS no-execute: doesn't protect
Canary: doesn't protect

Heartbleed is a buffer-overrun vulnerability, but it's a "read-only" attack!

It's not code-injection, it's not return-to-libc.

"Solution:" adjust C with array-bounds checks

There have been a dozen or more language designs like this. None have ever caught on. The problem is, then it's really not C any more.

“Solution:” Java, C#, etc.

Type-safe languages with array-bounds checking and garbage collection . . .

Actually, that **is** the solution.

Language choice as an ethical issue?

From a software engineering ethics point of view:

If you deliberately choose an unsafe programming language, there had better be a justified reason.

If you carelessly choose an unsafe programming language, then you're being unethical.

Agenda

Famous bugs

Common bugs

Testing (from lecture 6)

Reasoning about programs

Ethics of performance tuning

Tune your program (1950-2050)

samples	%	image name	app name	symbol name
20871	75.8807	libc-2.17.so	buzz1	__strcmp_sse42
5732	20.8398	buzz1	buzz1	SymTable_get
257	0.9344	buzz1	buzz1	SymTable_put
256	0.9307	buzz1	buzz1	sortCounts
105	0.3817	[illegible]	[illegible]	[illegible]
92	0.3345	no-[illegible]	[illegible]	/no-vmlin[illegible]
75	0.2727	lib[illegible]	[illegible]	fgetc
73	0.2654	libc[illegible]	[illegible]	__strlen_sse2_pmi[illegible]
[illegible]	0.0364	buzz1	buzz1	readInput
[illegible]	0.0327	libc-2.17.so	buzz1	__ctype_tolower_lo[illegible]
[illegible]	[illegible]0291	libc-2.17.so	buzz1	_int_malloc
[illegible]	[illegible].0109	libc-2.17.so	buzz1	__ctype_b_loc
[illegible]	0.0109	libc-2.17.so	buzz1	malloc
[illegible]	0.00[illegible]	libc-2.17.so	buzz1	__strcpy_sse2_unaligned
[illegible]	[illegible]036	buzz1	buzz1	SymTable_map
[illegible]	[illegible].0036	ld-2.17.so	time	bsearch
[illegible]	[illegible].0036	libc-2.17.so	buzz1	malloc_consolidate
[illegible]	0.0036	libc-2.17.so	buzz1	strcpy
1	0.0036	libc-2.17.so	time	__write_nocancel

Name of the function

% of execution time spent in this function

Name of the executable program

Name of the binary executable

Name of the running program

BUSINESS DAY

The New York Times

VW Is Said to Cheat on Diesel Emissions; U.S. to Order Big Recall

By CORAL DAVENPORT and JACK EWING SEPT. 18, 2015

WASHINGTON — The Obama administration on Friday directed Volkswagen to recall nearly a half-million cars, saying the automaker illegally installed software in its diesel-power cars to evade standards for reducing smog.

illegally installed software

General principle of extreme performance tuning

Steering wheel never moves?

In the test harness

Run the NO_X trap

(uses more gas,

wears out the

NO_X trap)

Not in the test harness

Turn off the

NO_X trap

(great gas mileage,

but unfortunately,

40x more nitrous-oxide pollution)

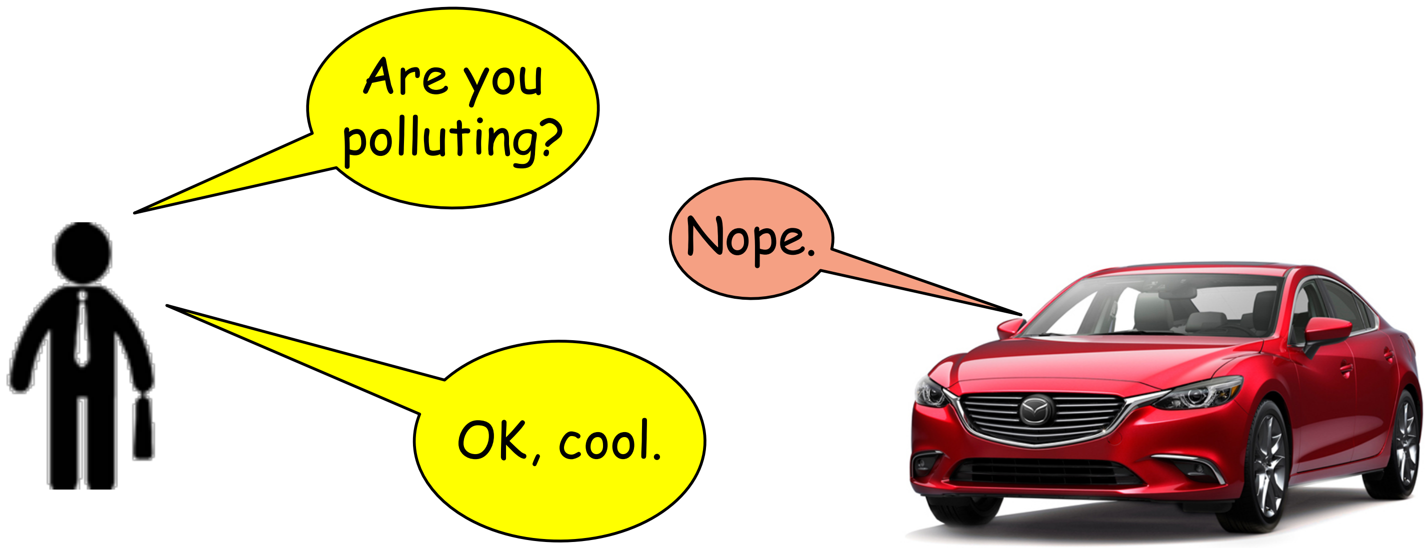
Real-life NJ DMV test harness

New style (in many states) DMV emissions testing
for cars made since 1996

How the test harness works

Programming challenge

Write a program that cheats on this test:



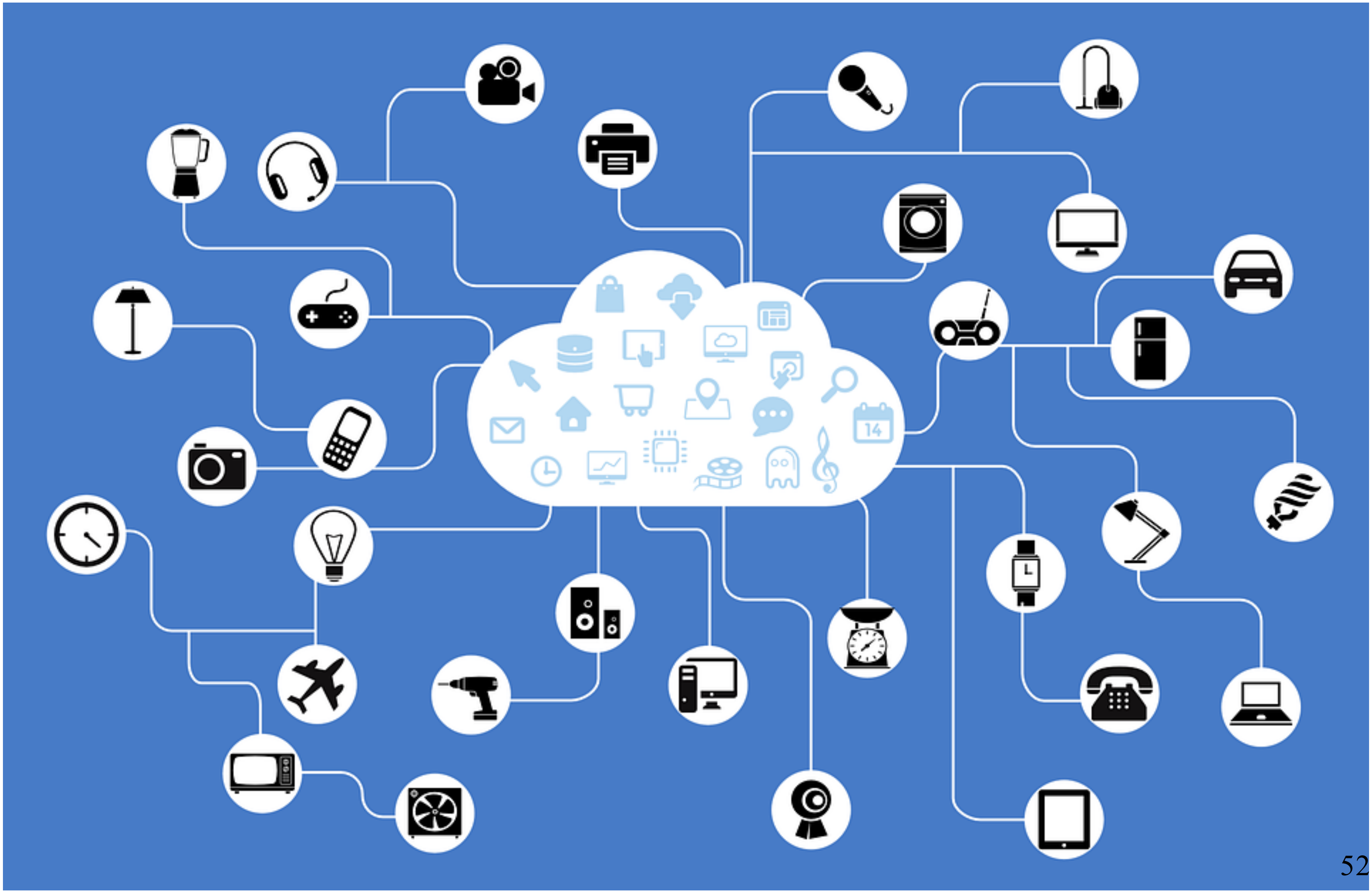
Solution:

printf("Nope.");

Obviously trivial! Therefore we rely on law and ethics to prevent this cheating.

What if you didn’t cheat <u>on purpose</u>?

The Internet of Things



KrebsonSecurity

In-depth security news and investigation

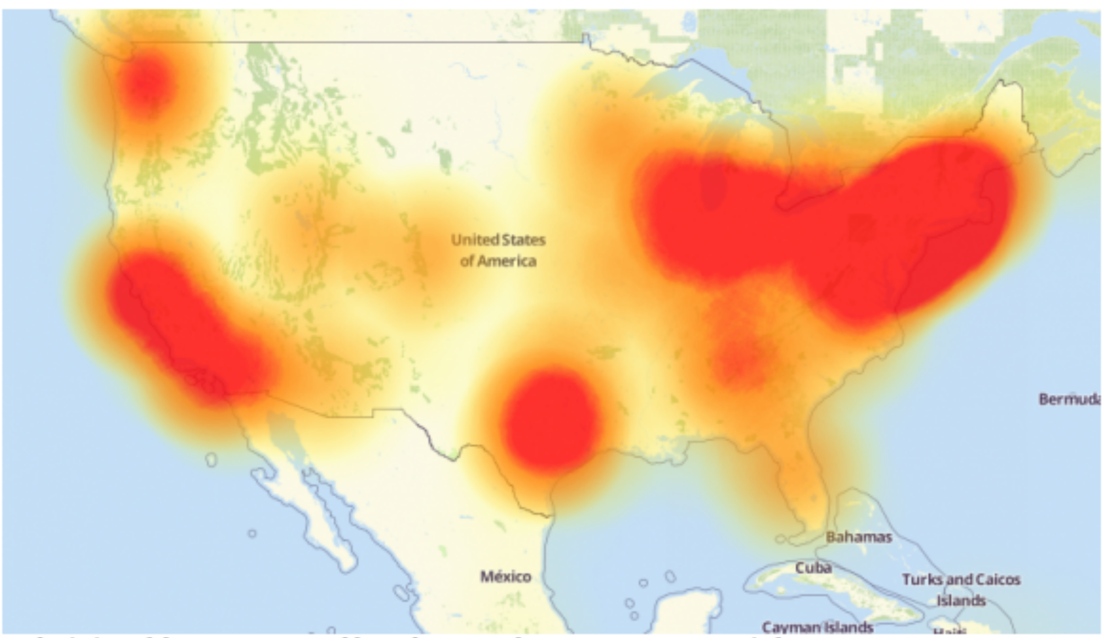
21 OCT 16

Hacked Cameras, DVRs Powered Today's Massive Internet Outage

October 21, 2016

A massive and sustained Internet attack that has caused outages and network congestion today for a large number of Web sites was launched with the help of hacked "Internet of Things" (IoT) devices, such as CCTV video cameras and digital video recorders, new data suggests.

Earlier today cyber criminals began training their attack cannons on **Dyn**, an Internet infrastructure company that provides critical technology services to some of the Internet's top destinations. The attack began creating problems for Internet users reaching an array of sites, including Twitter, Amazon, Tumblr, Reddit, Spotify and Netflix.



A depiction of the outages caused by today's attacks on Dyn, an Internet infrastructure company. Source: Downdetector.com.

The Internet of Things

Manufacturer A sells a "thing" (wifi router, toaster, thermostat, baby monitor, coffee maker, fitbit, football helmet, ...) for $50,

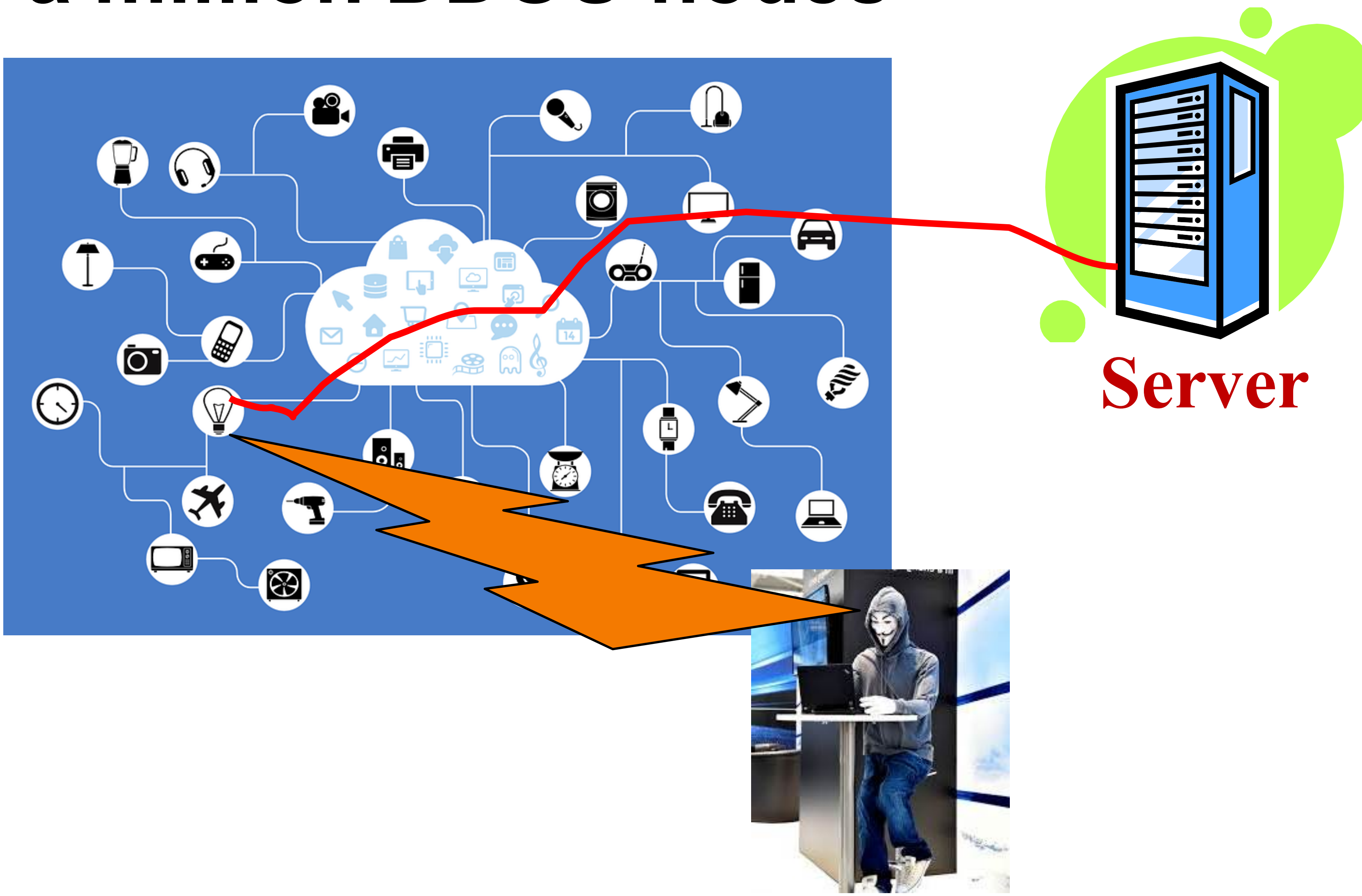
. . . full of security vulnerabilities (buffer overruns, SQL injection, etc ...)

Manufacturer B pays their engineers to spend a few more days, be a bit more careful, sells the "thing" for $51.

The Internet of Things

Consumer can't tell the difference, might as well buy the cheaper one

Hack a million devices, gain a million DDOS nodes



Does carelessness pay?

Fixing the "IoT security problem" is an open problem, from a regulatory point of view.

From a software engineering ethics point of view:

Your bug may harm the entire Internet.

Don't make and sell stupidly insecure devices!

The Rest of the Course

Assignment 7

- Due on Dean's Date (May 16) at 5 PM
- Cannot submit past 11:59 PM
- Can use late pass (but only until 11:59 PM)

Office hours and exam prep sessions

- Will be announced on Piazza

Final exam

- When: Friday 5/19, 1:30 PM – 4:30 PM
- Where: Friend Center 101
- Closed book, closed notes, no electronic devices

Course Summary

We have covered:

Programming in the large

- The C programming language
- Testing
- Building
- Debugging
- Program & programming style
- Data structures
- Modularity
- Performance

Course Summary

We have covered (cont.):

Under the hood

- Number systems
- Language levels tour
  - Assembly language
  - Machine language
  - Assemblers and linkers
- Service levels tour
  - Exceptions and processes
  - Storage management
  - Dynamic memory management
  - Process management
  - I/O management
  - Signals

Thank you!