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

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

• mission-critical systems

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)
fixed 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

Specification: General Program Checker

Right or Wrong

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

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

Reasoning about Programs

Example:

```
int factorial(int x) {
  int y = 1;
  int z = 0;
  while (z != x) {
    z = z + 1;
    y = y * z;
  }
  return y;
}
```

- 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, y=1, z=1
- At line 4, after 2nd iteration of loop: x=3, y=2, z=2
- At line 4, after 3rd iteration of loop: x=3, y=6, z=3
- Since z == x, exit loop, return 6: It works!
Reasoning about Programs

Example:

```c
int factorial(int x) {
  int y = 1;
  while (x != 0) {
    y = y * x;
    x = x - 1;
  }
  return y;
}
```

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=3, y=1, z=1
- At line 4, after 2nd iteration of loop: x=2, y=2, z=2
- At line 4, after 3rd iteration of loop: x=1, y=6, z=3
- At line 4, after 4th iteration of loop: x=0, y=24, z=4
- Since z == x, exit loop, return 24: It works!

Reasoning about Programs

Example:

```c
int factorial(int x) {
  int y = 1;
  while (x != 0) {
    y = y * x;
    x = x - 1;
  }
  return y;
}
```

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=999, y=1, z=1
- At line 4, after 2nd iteration of loop: x=998, y=2, z=2
- At line 4, after 3rd iteration of loop: x=997, y=6, z=3
- At line 4, after 4th iteration of loop: x=996, y=24, z=4

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 = \( 1 + 2 + 3 + ... + k + (k+1) \) = \( k * (k+1)/2 + (k+1) \)
  - Using the inductive hypothesis
  - \( = (k* (k+1)/2) + 2(k+1) \)
  - \( = (k+1)(k+2)/2 \)
  -Therefore, claim is true for all \( n \)

Reasoning about Programs

Example:

```c
int factorial(int x) {
  int y = 1;
  while (x != 0) {
    y = y * x;
    x = x - 1;
  }
  return y;
}
```

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=999, y=1, z=1
- At line 4, after 2nd iteration of loop: x=998, y=2, z=2
- At line 4, after 3rd iteration of loop: x=997, y=6, z=3
- At line 4, after 4th iteration of loop: x=996, y=24, z=4

Reasoning about Programs

Example:

```c
int factorial(int x) {
  int y = 1;
  while (x != 0) {
    y = y * x;
    x = x - 1;
  }
  return y;
}
```

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=999, y=1, z=1
- At line 4, after 2nd iteration of loop: x=998, y=2, z=2
- At line 4, after 3rd iteration of loop: x=997, y=6, z=3
- At line 4, after 4th iteration of loop: x=996, y=24, z=4

Loop Invariant

Example:

```c
int factorial(int x) {
  int y = 1;
  while (x != 0) {
    y = y * x;
    x = x - 1;
  }
  return y;
}
```

Try to prove by induction that the loop invariant holds
- Loop invariant (assertion at line 4): \( y = f(x) \)
- Try to prove by induction that the loop invariant holds
- Use induction over \( n \), the number of loop iterations
- Inductive hypothesis: Assume that claim is true for \( n=k \)
- 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 = \( 1 + 2 + 3 + ... + k + (k+1) \) = \( k * (k+1)/2 + (k+1) \)
  - Using the inductive hypothesis
  - \( = (k* (k+1)/2) + 2(k+1) \)
  - \( = (k+1)(k+2)/2 \)
  - Therefore, at line 4, \( y' = y \) (update values)
- Want to keep going on???
Proof of Correctness

We have proved the loop invariant (assertion at line 4): \( y = \text{fac}(z) \)

What should we do now?
- Case analysis on loop condition
  - If loop condition is true, i.e., if \( z \neq x \), execute loop again, \( y = \text{fac}(z) \)
  - If loop condition is false, i.e., if \( z = x \), exit the loop
  - At line 8, we have \( y = \text{fac}(z) \) AND \( z = x \), i.e., \( y = \text{fac}(x) \)
  - Thus, at return, \( y = \text{fac}(x) \)

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
- 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 \geq 0\} y = \text{factorial}(x); \{y = \text{fac}(x)\}

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)!

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

F-Soft Model Checker

Model Checking

- Temporal logic
  - Used for specifying correctness properties
    - [Pruel, 1977]
  - Model checking
    - Turing Award 2007
    - Verifying temporal logic properties by state space exploration
      - [Clarke & Emerson, 1981] and [Quelle & Sifakis, 1981]

Counterexample Trace

Array 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

---

1972

Niklaus Wirth designs Pascal language,
with supposedly ironclad array-bounds checking.

Turing award 1984

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
eruns 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.)

---

Cat-and-mouse
regarding
the buffer overrun problem
**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!

---

**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!

---

**"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!)

---

**"Solution:" hardware permissions**

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

(required inventing new hardware mechanism!)

---

**"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

---

**Buffer overrun with random stack-start**

a.out

What is your name?

Segmentation violation

BUT:

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

---

Buffer overrun with random stack-start

a.out

What is your name?

Segmentation violation

BUT:

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

---

**"Solution:" canary values**

"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!)

---

**"Solution:" more hardware permissions**

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

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

Tune your program (1950-2050)

General principle of extreme performance tuning

Steering wheel never moves

In the test harness
- Run the NOₓ trap (uses more gas, wears out the NOₓ trap)

Not in the test harness
- Turn off the NOₓ 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

Are you polluting?

Nope.

OK, cool.

Programming challenge

Write a program that cheats on this test:

Are you polluting?

Nope.

OK, cool.

Solution:

printf("Nope.");

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

What if you didn’t cheat on purpose?

The Internet of Things

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

Server

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!