Last time: performance (QoS - quality of service)

This class: correctness

Correctness is always important (compilers, query optimizers, etc.)

have program Q and convert to Q'

Q: orig

Q': modified

what if there isn't 2 programs to check equivalence? what if only P?

prove they're equivalent

$Q \equiv Q'$

(reducible to Halting problem, but at least if test 1mil times, can say they're behaviorally equivalent)

need complete spec (usually written in logic)

2 kinds of correctness:

① total: assumes there is a spec and you try to show P is behaviorally equivalent to spec

stronger

$P \equiv spec$ ($P \subseteq spec$ really)

$\forall i : O(s) \equiv O(p)$

O: output

② partial:

ex: program never derefs null

(any program that does deref null is wrong)

ex: no uninitialized vars

ex: use after free (C/C++ problem)

ex: buffer overflow

ex: exceptions (e.g. file not found error)

ex: segv

rules that should be True

weaker

tools that look for these are error checkers / linters

We assume that any correct program does not break these rules

implicit spec for programs

ex:
```
if (x = 0) {
    :
}
```
← assignment, so will always fall through

linters (pattern checkers) would report bad programming practices

disadv of linters?

- 1000s of warning messages, referred to as false positives
- also suffers from false negatives (if bug is not part of pattern)

precision: inverse of fp, report ⇒ bug

recall: bug ⇒ report

perfect recall:

```
for (i=0; i<n; i++) {
```

if there's a bug, def report

cout << i << "has a bug" << endl

}

perfect precision:

cout << "no bugs" << endl

precision-recall tradeoff

ppl now prefer high precision in general

The Spec Problem

- writing spec is hard
- spec is a program just in another lang → spec is long + complicated too
- is the spec correct? we don't know...
- who guards the guardians?
- Program is the spec (there's no real spec)
  - paper: Bugs as Deviant Behavior
    - programs act weird sometimes means probably a bug
    - static v. dynamic analysis
      - ↙ looks at code
      - ↘ looks at execution

Coverity: static analysis tool

collect factual info by running program in abstract interpretation (execute symbolically)

- could have infeasible paths in program, but static analysis does not know about these
- state space explosion: static analysis needs to control this

ex: $x \geq 12$, $x \leq 130$ } many states between start + end

losing precision

:

also lose precision w/ loops

ex:
```
while (true) {
  x = x+1
}
```

all we know is $x \geq x'$ (↑ orig x)

must analyses v. may analyses

false pos example:

index ≥ 0 index ≤ 1 mil

a = new int [1000]

a[index] buffer overflow?

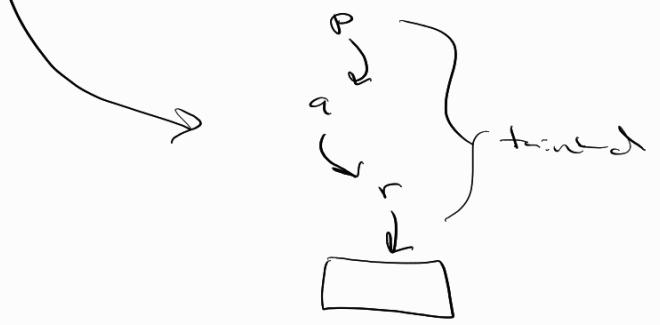
coverity ranks errors!

- ranked on state space, precision, etc
- better than giving someone 1mil errors and saying good luck

some other static analysis tools focus on security over correctness:

- exploitable bug ⇒ $$$
- security-based analysis
- classic bugs:

- use after free
- tainted (transitive property)
- writing to a file (e.g. overwrite password file)



information flow: prove absence of tainted property

Dynamic Analysis:

- Valgrind
  - shadow memory: version of the mem (identical or compact representation)
    - uninitialized values → 0 or 1
    - malloc + free mem → M or F
  - precision super high
  - if no error found, then particular execution path doesn't have error
    - generally do not generalize! (static can)
    - not much recall

	Dynamic	Static
P	✓	☹
R	☹	✓

- valgrind uses memcheck, cachegrind, helgrind (race detection)
- Google has ASan, UBSan, + TSan

rather than ASan (overhead), typically use logging + telemtry (watson)

↳ usually get stacktraces (not helpful)

wish we had MRE (minimal reproducible example?)

PII

GDPR (rules for managing + collecting PII)

↳ makes complicated to find bugs

So nothings finding all bugs...

Do better testing

If ppl don't write specs, will they write tests?

- unit tests
- assertions
- regression tests

} add to continuous integration

As # of tests grows, cost grows (time btwn commits grows)

Also more complicated tests:

can't anticipate all

- end to end tests

anticipate

ppl hate writing tests, and written tests aren't great (corner case?)

- developers v. testers (moral hazard)

↓

if testers are here, devs will write worst code (this actually was true?)

- coverage: need tests to increase coverage
  - line
  - branch
  - full path

- flakey tests: heisenbug in tests (says test failed)

- fuzzing:
  - discovered by accident (line noise ex)
  - all Unix tools crashed on random inputs
  - no are anticipated randomness!
  - easy to generate randomness, but not random structured inputs (ex. JSONs)
  - we now have great fuzzers
  - these have found tons of bugs
  - 2 ways to work:
    - through static analysis
      - DART: concrete + symbolic analysis

```
input
if (c1) {

}
else {

}
```

→ SAT Solver (NP complete) — avg case of SAT solvers are fast! (eg Z3)

synthesize a concrete state + include random input

AFL is another fuzzer

- American Fuzzy Lop
- one of the most effective fuzzers
- mutates inputs to see if any further coverage
- uses genetic algorithms (really a giant hack but works great!)
- parallizable, very quick
- very successful @ finding security bugs

Thursday Exam!

Last wk is for final project (Juan taking attendance)