Lecture 5

dinsdag 29 november 2022 10:36

1. static code analysis —(major theme)→ halting problem forces us to give up some goals since we cannot perfectly prove security
   ↓
   simplifications and frameworks
2. fuzzing (generation + execution of tests)
   also: dynamic code analysis ↳ choose smart inputs to try and find errors
   ↳ at runtime!
3. review by experts/penetration testing
   ↳ expensive

1. automatic
   computationally cheap
   false alarms e.g. false positives
   covers code which is rarely executed (i.e. cornercases)
   can find the 'origin' of the problem relatively easily

---

2. cannot raise false alarms
   also works without source code
   certain parts of the code will rarely be executed
   origin of problem is difficult to find

---

3. does not scale well

---

all methods do not find all bugs
especially context-/environment-related bugs

$$\text{precision} = \frac{\text{true positives}}{\text{rejected}} \qquad \text{recall} = \frac{\text{true positives}}{\text{bad}}$$

(rejected = positives; bad = true positive + false negative)

$$F\text{-measure} = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = \frac{2 \cdot \text{recall} \cdot \text{precision}}{\text{recall} + \text{precision}}$$

$$F_\beta\text{-measure} = \frac{(1+\beta^2)\ \text{recall} \cdot \text{precision}}{\text{recall} + \beta^2 \cdot \text{precision}}$$

β higher → recall more important than precision

complete : all bad programs are recognized as bad
sound : only bad programs are recognized as bad

security is measured relative to some formal/precise statement
↓
more precise ⇒ easier to automate

history-based, state-based, transition-based, functional, operational } specification

safety properties → program will never reach a bad state
(assertions, type checks)

liveness properties → program will eventually reach a bad state

we cannot have all of
- completeness → if recognized as bad ⇒ code is bad; → often given up (no false positives)
- soundness (no false negatives)
- automated system → find all bad programs
- power of expression

static code analysis
often searches for patterns or code fragments that meet some rules

techniques:
- control flow analysis
- data flow analysis
- pattern checking for compliance with coding rules/standards
- type systems
- instrumentation of binary code e.g. add additional checks to compiled binary (debug, e.g. memcheck/valgrind)

call graph : which function call which ones?
CG — edges = function call

control flow graph : considers all paths that control flow can take in one function
CFG

inter-procedural control flow graph → inserts CFGs into CG

static code graph → undecidable
dynamic code graph → for a specific run only

binary code instrumentation
↓
- step-wise execution
- breakpoints: halt at points
- watchpoints: give feedback (and possibly: halt) when conditions are met

valgrind : for finding memory leaks,
but 20-30 times slower (only used in testing)
↓
things like this make e.g. Java slow

program verification tries to approximate all possible behaviors
often such that if there is a bug in the original, there is one in the approximation as well

techniques
- model checking
- deductive verification
- abstract interpretation keyword: template

WORK OUT in more detail