CS 242

Code analysis tools

John Mitchell

Announcements

- Midterm exam
  - Wed Oct 24, 7-9 PM Gates B01 (next door!)
  - EE282 students – take alternate EE282 exam
- Homework 4
  - Posted on web today
  - Will not be graded – solutions in class next Monday

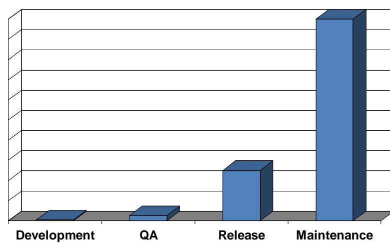
Outline

- Motivation for tools
  - Cost of serious bugs
  - Time-to-market issues
- Some basic examples
  - Purify
- Abstract interpretation
- Satisfiability-based analysis

Serious cost of software errors

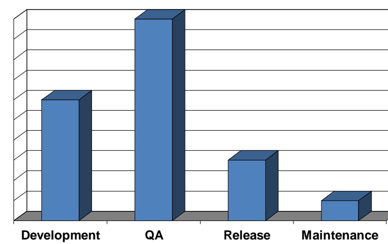
- NASA Mariner 1 (July 22, 1962)
  - Off-course during launch, missing 'bar' in its FORTRAN software
- Therac-25 (1985-1987)
  - Six accidents involved massive radiation overdoses to patients
  - Software errors combined with other problems
- AT&T long distance network crash (Jan 15, 1990)
  - Missing break in C switch statement
- Patriot MIM-104 (Feb 25, 1991)
  - Iraqi Scud hit barracks in Dharan, Saudi Arabia, killing 28 US soldiers
  - Failure to intercept Scud caused by software error related to clock skew
- Ariane 5 Bug crash (June 4, 1995)
  - Software errors in inertial reference system, cost: $7.5 billion
- Mars Orbiter (Sept 1999)
  - Crashed because of wrong units in a program
- Mars Rover (Jan 21, 2004)
  - Freezes due to too many open files in flash memory

Cost of Fixing a Defect



Credit: Andy Chou, Coverity

Number of Bugs Found



Credit: Andy Chou, Coverity

IT Economics (1)

- The first distinguishing characteristic of many IT product and service markets is network effects
  - Metcalfe's law – the value of a network is the square of the number of users
  - Real networks – phones, fax, email
  - Virtual networks – PC architecture versus MAC, or Symbian versus WinCE
- Network effects tend to lead to dominant firm markets where the winner takes all

Credit: Ross Anderson

IT Economics (2)

- Second common feature of IT product and service markets is high fixed costs and low marginal costs
  - Competition can drive down prices to marginal cost of production
  - This can make it hard to recover capital investment, unless stopped by patent, brand, compatibility …
- These effects can also lead to dominant-firm market structures

Credit: Ross Anderson

IT Economics (3)

- Third common feature of IT markets is that switching from one product or service to another is expensive
  - E.g. switching from Windows to Linux means retraining staff, rewriting apps
  - Shapiro-Varian theorem: the net present value of a software company is the total switching costs
- This is why so much effort is starting to go into accessory control – manage the switching costs in your favor

Credit: Ross Anderson

IT Economics: Conclusion

- High fixed/low marginal costs, network effects and switching costs all tend to lead to dominant-firm markets with big first-mover advantage
- So time-to-market is critical
- Microsoft philosophy of 'we'll ship it Tuesday and get it right by version 3' is not perverse behaviour by Bill Gates but driven by economics
- Whichever company had won in the PC OS business would have done the same

Credit: Ross Anderson

What can tools do for you?

- Impractical to prove program correctness
- Run-time instrumentation for better testing
  - Purify – detect memory errors
  - Race-condition detectors
- Static analysis to find specific problems
  - Find systematic bugs that are hard to find otherwise
    - Null pointer dereference
    - Protocol errors: open file, read/write, close
    - Bad input checking and buffer overflow
    - Exceptional conditions: divide by zero, int/float overflow
  - State-of-the-art tools are effective and improving

Purify

- Goal
  - Instrument program to find memory errors
    - Out-of-bounds: access to unallocated memory
    - Use-before-init
    - Memory leaks
- Technique
  - Works on relocatable object code
    - Link to modified malloc that provides tracking tables
    - Tracks *heap* locations only
  - Memory access errors: insert instruction sequence before each load and store instruction
  - Memory leaks: GC algorithm

Current static analysis tools

- Commercial products

- Public distribution with commercial sponsors

- Microsoft: PREfix, PREfast, …
  - Some tools available in Visual Studio

Terminology for static analysis

- Sound
  - If tool reports "no bugs," then program has no bugs
- Complete
  - If program has no bugs, tool will report "no bugs"
- Tradition in software verification
  - Program has no bugs if no program execution exhibits specific error (e.g., null pointer deference) at run time
  - Halting problem => no tool that halts is sound and complete
- Modern bug-finding tools
  - "Unsound" – do not find all bugs, but try to report as many meaningful bugs as possible
  - Creative engineering based on extensive experience

Abstract Interpretation

- Basic idea
  - Compute denotational semantics of program
  - Using nonstandard interpretation with finite domains
- Halting problem?
  - If there are only finitely many values, for finitely many variables, then cannot loop forever
- Additional theory
  - If abstract domain (finite representation of program properties) is related to standard domain in certain way, then this method is sound (but not complete).

Expressions, revisited

- Syntax

  d ::= 0 | 1 | 2 | … | 9

  n ::= d | -d | nd

  e ::= x | n | e * e | e + e
- Semantics value E : exp x state -> numbers

  E [[0]] = 0 E [[1]] = 1 …

  E [[-d]] s = - E[[d]] s

  E [[nd]] s = 10*E[[n]] s + E[[d]] s

  E [[e_1 * e_2]] s = E[[e_1]] s * E[[e_2]] s

  E [[e_1 + e_2]] s = E[[e_1]] s + E[[e_2]] s

Expressions: computing sign

- Syntax

  d ::= 0 | 1 | 2 | … | 9

  n ::= d | -d | nd

  e ::= x | n | e * e | e + e
- Semantics E : exp x state -> {-,0,+,± }

  E [[0]] = 0 E [[1]] = + …

  E [[-d]] s = - E[[d]] s where -- =+, -0=0, -+=-, -±=±

  E [[e_1 * e_2]] s = E[[e_1]] s * E[[e_2]] s

  where -*- =+, -*0=0, -*+=+, -*±=±, …

  E [[e_1 + e_2]] s = E[[e_1]] s + E[[e_2]] s

  where -+- =-, -+0=0, -++=±, …

Is this interpretation "right"?

- Meaning of abstract values
  - Each abstract value {-,0,+,± } corresponds to set

    - ≈ {n | n<0} 0 ≈ {n | n<0}

    + ≈ {n | n>0} ± ≈ N
  - Operations on sets respect this correspondence

    E [[e_1 * e_2]] s = E[[e_1]] s * E[[e_2]] s

    where -*- =+, -*0=0, -*+=+, -*±=±, …
  - We need ± because there is not always enough information to give -,0,+

Example: uninitialized variables

- Possible values for variables
  - State is either s: variables -> { OK, NOK} or *wrong*
  - E [[x]] s = s(x) if s not wrong
  - E [[e_1 + e_2]]s = if E[[e_1]]s = OK and E[[e_2]]s= OK then OK else NOK
- Meaning of program
  - C[[P]]s is either an updated state or *wrong*
  - C[[x := e]]s = if E[[e]]s = OK then s' with s'(x)=OK else *wrong*

Conditional

- Wrong if either branch is wrong

  C[[if B then P else Q]]s =

  if E[[B]]s = OK then worst(C[[P]]s, C[[Q]]s)

  else *wrong*

  where worst(s_1,s_2)(x) = if s_1(x)=s_2(x)=OK then OK

  else NOK
- How could we do better?
  - Track more values
  - State : Variables -> {0, 1, 2, 3, 4, … 99, OK, NOK}

Tracking values of variables

- Avoid false paths

  x := 0;

  y := x+2;

  if x=0 then z := x+1 else x := x+1;

  y := z;
- How does this work

  C[[if B then P else Q]]s :

  E[[B]]s = true ⇒ C[[P]]s

  E[[B]]s = false ⇒ C[[Q]]s

  E[[B]]s = OK ⇒ worst(C[[P]]s, C[[Q]]s)

  E[[B]]s = NOK ⇒ *wrong*

Loops

C[[while B do P]] = the function f such that

f(s) = if E[[B]]s then f(C[[P]](s)) else s

Solution involves unions and least upper bounds

- Calculation for finite domains

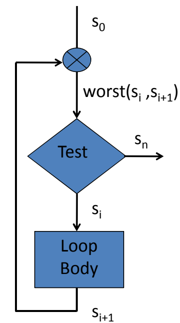
  f(s_0) = if E[[B]]s_0 is false then s_0

  else if E[[B]]s_2 is false then s_2

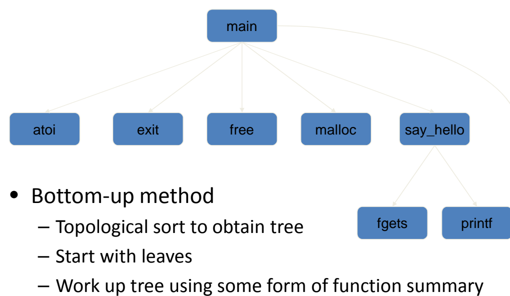
  else if E[[B]]s_4 is false then s_4

  else …. (stop if s_{i+2} = s_i)

  where s_{i+2} = worst(s_i , C[[P]]s_i)



Interprocedural analysis

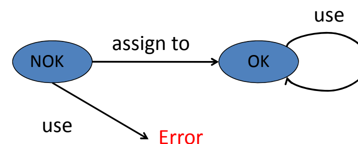


- Bottom-up method
  - Topological sort to obtain tree
  - Start with leaves
  - Work up tree using some form of function summary

Figure: Andy Chou, Coverity

State diagrams for abstract values

- Uninitialized variables



  - Abstract interpretation can be used to track information about state of each variable
  - Report program error when a variables is misused

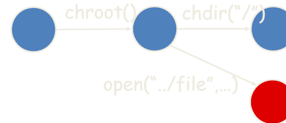
Example Java null ptr bugs

```
//com.sun.corba.se.impl.naming.cosnaming.NamingContextImpl
  if (name != null || name.length > 0)

//com.sun.xml.internal.ws.wsdl.parser.RuntimeWSDLParser
  if (part == null | part.equals(""))

// sun.awt.x11.ScrollPanePeer
  if (g != null)
      paintScrollBars(g,colors);
  g.dispose();
```

Credit: W. Pugh

Chroot checker

- chroot() changes filesystem root for a process
  - Confine process to a “jail” on the filesystem
- Doesn’t change current working directory



Credit: Andy Chou, Coverity

Many bugs to detect

- Some examples

- Crash Causing Defects
- Null pointer dereference
- Use after free
- Double free
- Array indexing errors
- Mismatched array new/delete
- Potential stack overrun
- Potential heap overrun
- Return pointers to local variables
- Logically inconsistent code
- Uninitialized variables
- Invalid use of negative values
- Passing large parameters by value
- Underallocations of dynamic data
- Memory leaks
- File handle leaks
- Network resource leaks
- Unused values
- Unhandled return codes
- Use of invalid iterators

Credit: Andy Chou, Coverity

Tainting checkers

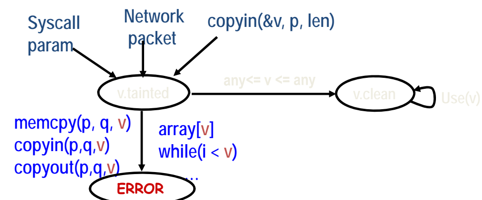
Credit: Andy Chou, Coverity

Application to Security Bugs

- Stanford research project
  - Ken Ashcraft and Dawson Engler, Using Programmer-Written Compiler Extensions to Catch Security Holes, IEEE Security and Privacy 2002
  - Used modified compiler to find over 100 security holes in Linux and BSD
  - http://www.stanford.edu/~engler/
- Benefit
  - Capture recommended practices, known to experts, in tool available to all

Sanitize integers before use

Warn when unchecked integers from untrusted sources reach trusting sinks



Linux: 125 errors, 24 false; BSD: 12 errors, 4 false

Example security holes

- Remote exploit, no checks

```
/* 2.4.9/drivers/isdn/act2000/capi.c:actcapi_dispatch */
isdn_ctrl cmd;
...
while ((skb = skb_dequeue(&card->rcvq))) {
   msg = skb->data;
   ...
   memcpy(cmd.parm.setup.phone,
          msg->msg.connect_ind.addr.num,
          msg->msg.connect_ind.addr.len - 1);
```

Example security holes

- Missed lower-bound check:

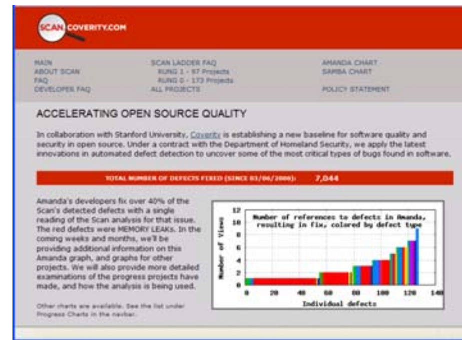
```
/* 2.4.5/drivers/char/drm/i810_dma.c */

if(copy_from_user(&d, arg, sizeof(arg)))
   return -EFAULT;
if(d.idx > dma->buf_count)
   return -EINVAL;
buf = dma->buflist[d.idx];
Copy_from_user(buf_priv->virtual, d.address, d.used);
```

Results for BSD and Linux

- All bugs released to implementers; most serious fixed

Violation	Linux Bug	Linux Fixed	BSD Bug	BSD Fixed
Gain control of system	18	15	3	3
Corrupt memory	43	17	2	2
Read arbitrary memory	19	14	7	7
Denial of service	17	5	0	0
Minor	28	1	0	0
Total	125	52	12	12

Bugs in open-source software



False Positives

- What is a bug?
  - Coverity: Something the user will fix.
- Many sources of false positives
  - False paths
  - Execution environment assumptions
  - …
- What's an acceptable level?
- How can this be achieved?

User-pointer inference

- Problem: which are the user pointers?
  - Hard to determine by dataflow analysis
  - Easy to tell if kernel *believes* pointer is from user!
- Belief inference
  - "*p" implies safe kernel pointer
  - "copyin(p)/copyout(p)" implies dangerous user ptr
  - Error: pointer p has both beliefs.
- Implementation: 2 pass checker
  - inter-procedural: compute all tainted pointers
  - local pass to check that they are not dereferenced

Microsoft Tools

- PREfix
  - detailed, path-by-path interprocedural analysis
  - heuristic (unsound, incomplete)
  - expensive (4 days on Windows)
  - primarily language usage defects
- PREfast
  - simple plug-ins find defects by examining functions' ASTs
  - desktop use
  - easily customized
- Widely deployed in Microsoft
  - 1/8 of defects fixed in Windows Server 2003 found by these tools

[Larus ASPLOS]

SAT-based tools

- Satisfiability
  - General method for "solving" Boolean formulas
- Conversion of code to satisfiability
  - Loop unrolling – wont talk about this
  - Single-assignment form
  - Generate formula characterizing error conditions
  - Use SAT solver to find error runs
- Applications
  - Alternate method for many kinds of errors
  - Track actual program values: fewer false errors,

Satisfiability

- Propositional formulas
  - $(A \lor B \lor \neg C) \land (A \lor \neg B \lor C) \land (\neg A \lor B \lor C) \land \ldots$

    At least two of A,B,C must be true
- Satisfiability
  - Find assignment of True/False to propositional variables that makes formula true
- Classical NP-complete problem
  - An algorithm that solves SAT efficiently can be used to solve many other *hard* problems efficiently

Conversion to passive form

- Change imperative program to form where each variable assigned ≤ once on each path (pure functional program)

if (c != 0)	if (c_0 != 0)
v = v + 1	$v_1 = v_0 + 1$
v = v * 2	$v_2 = v_1 * 2$
else	else
v = v -1	$v_1 = v_0 - 1$
	$v_2 = v_1$
v = v + 1	$v_3 = v_2 + 1$

- Why?
  - Name value of each variable at each program point
  - This form (or similar) also useful in compiler optimization

Formula characterizing error

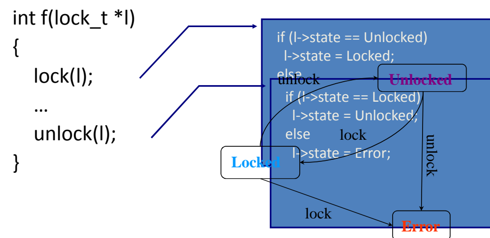
- Extract formula, using new variables, for error condition

		← $(\neg c_0 \land v_0<3.5) \lor (c_0 \land v_0<10)$
if (c != 0)	if (c_0 != 0)	
v = v + 1	$v_1 = v_0 + 1$	← $v_0 < 3.5$
v = v * 2	$v_2 = v_1 * 2$	← $v_2 < 4.5$
else	else	
v = v -1	$v_1 = v_0 - 1$	← $v_0 < 10$
	$v_2 = v_1$	
v = v + 1	$v_3 = v_2 + 1$	← $v_2 < 9$
assert(v<10)	assert(v_3<10)	

- Easy way
  - $v_3<10 \land v_3 = v_2+1$
    $\land ((\neg c_0 \land v_1=v_0+1 \land v_2 = v_1*2) \lor (c_0 \land v_1=v_0-1 \land v_2=v_1))$
- Satisfy this formula => program trace with error

Example SAT tool: Saturn

```
int f(lock_t *l)
{
   lock(l);
   …
   unlock(l);
}
```



Slide credit: A Aiken

General FSM Checking

- Encode FSM in the program
  - State → Integer
  - Transition → Conditional Assignments
- Check code behavior
  - SAT queries

Slide credit: A Aiken

How are we doing so far?

- Precision: ☺
  - Bit-level analysis
  - And path-sensitive for every bit
- Scalability: ☹
  - SAT limit is 1M clauses
  - About 10 functions
- Solution:
  - Divide and conquer
  - Function summaries

Slide credit: A Aiken

FSM Function Summaries

- Summary representation (simplified):
  { P_{in}, P_{out}, R }
- User gives:
  - P_{in}: predicates on initial state
  - P_{out}: predicates on final state
  - Express interprocedural path sensitivity
- Saturn computes:
  - R: guarded state transitions
  - Used to simulate function behavior at call site

Slide credit: A Aiken

Lock Summary

```
int f(lock_t *l)
{
   lock(l);
   …
   if (err) return -1;
   …
   unlock(l);
   return 0;
}
```

- Output predicate:
  - P_{out} = { (retval == 0) }
- Summary (R):
  1. (retval == 0)
     *l: Unlocked → Unlocked
     Locked → Error
  2. ¬(retval == 0)
     *l: Unlocked → Locked
     Locked → Error

Slide credit: A Aiken

Lock checker for Linux

- Parameters:
  - States: { Locked, Unlocked, Error }
  - P_{in} = {}
  - P_{out} = { (retval == 0) }
- Experiment:
  - Linux Kernel 2.6.5: 4.8MLOC
  - ~40 lock/unlock/trylock primitives
  - 20 hours to analyze
    - 3.0GHz Pentium IV, 1GB memory

Slide credit: A Aiken

Bugs

Type	Bugs	False Pos.	% Bugs
Double Locking	134	99	57%
Ambiguous State	45	22	67%
Total	179	121	60%

Previous Work: MC (31), CQual (18), <20% Bugs

Slide credit: A Aiken

Conclusion

- Apply automated methods for error reduction
  - Software errors are expensive
  - Better software design and development environments can improve time to market
- Some basic tools are widely used
  - Purify – run-time instrumentation for memory errors
- Practical, informative static tools are now a reality
  - Abstract interpretation
    - Simulate execution using "abstract" domains of values
  - Satisfiability-based analysis
    - Leverage powerful SAT solvers to find errors, do other analysis
- Question: what language features lead to better tools?