

## Dynamic Analysis

## Homework 1

- Due Monday Oct 5, 9 AM on moodle
- On dynamic analysis (today's topic)
- Install and use an open-source tool: Daikon
- Add a very useful tool to your toolbox
- Understand how dynamic analysis works

Any questions?

## Today's plan

- Runtime monitoring
  - Rational Purify
- Dynamic invariant detection
  - Daikon

## Rational Purify

- IBM (formerly Rational, then bought by IBM)
- Memory debugging
  - uninitialized memory access
  - buffer overflow
  - improper freeing of memory
- Memory leak detection
  - memory blocks that no longer have a valid pointer

<http://www-01.ibm.com/software/awdtools/purify>

## The Problem (for Purify to solve)

- C/C++ are not type safe
- The compiler does not enforce type abstractions
- Does the runtime system?
  - no

### Memory

Memory

Object Object int[]

- What happens if we write here?

### The Problem (for Purify to solve)

- C/C++ are not type safe
- The compiler does not enforce type abstractions
- Does the runtime system?
  - no
- Possible to read or write outside of your intended data structure
- ... and many undesirable behaviors

### What can we do?

- Track each memory location

One of three states:

- Unallocated: cannot be read or written
- Allocated but uninitialized: cannot be read
- Allocated and initialized: can be read or written

### Memory

Memory

Object Object int[]

- What happens if we write here?

### Represent each byte's state with a machine

```

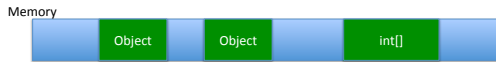
    graph TD
      Unallocated((Unallocated)) -- malloc --> Uninitialized((Uninitialized))
      Uninitialized -- write --> Initialized((Initialized))
      Initialized -- free --> Unallocated
    
```

Anything missing?

### How do we implement this?

- Keep a machine for each byte
- On each access
  - check the state of each byte
  - update the machine state
- Can instrument the binary (no need for source code)
  - Add code before each load and store
  - Represent the machines as a giant array
    - How many bits needed per byte of system memory?
- What's the overhead?
- Catches byte-level, but not bit-level errors
- Runtime CPU efficiency?
  - very slow, but worth it

## Memory



- We can detect errors in the blue areas.
- We can detect some errors in the green areas.
- But there are many others we cannot.

## Can we do better?

- We can detect unallocated or uninitialized accesses.
- Can we force all accesses to be that way?

## Padding between objects



If we disallow adjacent objects in memory (pad them), then all accesses past the end of an array access a blue zone

## Let memory age

- Do not allow reallocation of freed memory for some time
- Prevents errors caused by dangling pointers
- Both this and padding can be easily implemented in the malloc library

## Garbage collection

- Instead of bits, keep track of pointers to memory
- When no pointers are left, free the memory
- Where have we seen this before?

## In Practice

- These ideas work pretty well and are widely used.
- Often, it is OK to pay very high performance price to get system correctness.
- Dynamic analysis instruments the program, can maintain properties at runtime.

## Today's plan

- Runtime monitoring
  - Rational Purify
- Dynamic invariant detection
  - Daikon

## What is a program supposed to do?

- How do we know the program's specification?
- Maybe the developers wrote it down.
  - but often, that has errors
- Without a specification or some way to tell if behavior is correct, we cannot test!

## What is a specification?

- The documentation can be the specification
  - Informal
  - May contain mistakes
  - Can be hard to parse
- The program itself is a specification
  - Testing becomes a tautology
  - But is there some kind of testing this can facilitate?
    - Regression testing
  - Also great for program understanding, reasoning, etc.

## Use the program to find likely invariants

- Hypothesize an invariant
  - for example, `square(x) > 0`
- Run the program on many test inputs (without needing to know the outputs)
- If `square(x) > 0` in all the executions, it's a likely invariant.

## Example:

```
funny_sqrt(int x)
  bool positive = (x>0);
  if (positive)
    j = sqrt(x);
  else
    j = sqrt(-x);
  return j;
```

Test for  $-100 < x < 100$      $j \geq 0$

## What is an invariant? ( $j \geq 0$ is)

- Invariants hold at a program point
  - before a statement executes
  - after a statement executes
  - or maybe at all program points
- Invariants cannot reference variables out of scope
  - Is `j < abs(y)`?

### Can executions ever prove a property?

- Can show that a property holds in many executions.
- But can this method show that a property always holds?
- What can executions prove?
  - They can disprove invariants by finding an execution in which an invariant holds.

### Example: Is j always 0?

```
funny_sqrt(int x)
  bool positive = (x>0);
  if (positive)
    j = sqrt(x);
  else
    j = sqrt(-x);
  return j;
```

Test for  $-100 < x < 100$

No, when x is -100, j is 10.

j can be between 0 and 10

### How do we know if an invariant is likely?

Thesis:

Hypothesize *i* is an invariant at a program point.  
If many test cases do not disprove the hypothesis, conclude that *i* likely is an invariant.

This doesn't quite work...

### Example:

```
funny_sqrt(int x)
  bool positive = (x>0);
  if (positive)
    j = sqrt(x);
  else
    j = sqrt(-x);
  return j;
```

Test for  $0 < x < 100$

$x \geq j$   
positive = true

### What went wrong?

- We had many test cases  
none disproved the invariant
- But the hypothesis is not disproved because we didn't even execute the relevant line of code.

### Example:

```
funny_sqrt(int x)
  bool positive = (x>0);
  if (positive)
    j = sqrt(x);
  else
    j = sqrt(-x);
  return j;
```

Test for  $0 < x < 100$

$x \geq j$   
positive = true

### Solution: Use statistics!

- An invariant is only likely if
  - the observations do not disprove it
 AND
  - the **relevant** observations are statistically significant

### How to compute statistical significance?

- For a hypothesized invariant  $P(x,y)$   
What are the chances  $P(x,y)$  is satisfied under a random choice of  $x$  and  $y$ ?
- Assume  $0 \leq x, y < 1000$

$$\begin{aligned} P(x == y) &\approx .001 \\ P(x < y) &\approx .5 \\ P(x \neq y) &\approx .999 \end{aligned}$$

### What to compute

- We want a high confidence that invariants are not observed by chance
- The number of samples we need varies with the invariant
  - predicates have widely varying chances of being accidentally satisfied

### What can we do with unlikely invariants?

- If it is likely that a [non]invariant is an accident, don't report it.
- Give the user control of the confidence threshold.

An invariant may be true, but not be statistically significant when examined under some (all?) test suites.

### Which invariants do we check?

- Given a possible invariant, we can check if it is likely.
- But which possible invariants do we check?  
How many are there?

### How many are there?

- Ordering relationships over two variables:  
 $x < y, x == y, x > y, x \leq y, x \geq y, x \neq y$
- No problem. Just a finite number
  - If a program has  $n$  variables, how many possible such relationships are there?

$$\Theta(n^2)$$

### What about other types?

- $x = c$ , for some constant  $c$

many:  $2^{64}$  for ints on a 64-bit machine =  
18,446,744,073,709,551,616

### Use the computer for what it's good at

- Guess a HUGE number of possible invariants
- Check them all
- Only those that are likely true will survive
- Computers are great at this!

### So what do we do about the 18,446,744,073,709,551,616 ints?

Possible invariant:  $x=c$

- Don't store any at first.
- First time you see  $x$  assigned to some  $c$ , remember that  $c$ .
- Then check if  $x=c$  in all later executions.

### For others too

- Same idea for more-complex invariant types:
- For example:  
 $ax + b = y$
- **Two** observations of  $(x,y)$  is sufficient to solve for the only possible  $(a,b)$ .

### And others still

- We can do:
  - $\text{min}(\text{array})$
  - $\text{max}(\text{array})$
  - $\text{sum}(\text{array})$
  - etc.
- These expressions can be like variables:  
 $x = \text{min}(\text{array } z)$

### Review

- Guess lots of invariants
  - Check which ones hold
  - Keep statistics to check for statistical significance
- Those guesses that survived all the executions and are statistically significant are likely true.

Does not need expected execution outputs

### In practice

- This works!
- Finds interesting invariants for complex programs.
- Gives concise specifications
- Needs fewer executions than you'd think.

### False invariants die quickly



### Daikon

- Implements dynamic invariant detection
- Open source, free to use,
- Highly robust and customizable
- Takes some time to master but very powerful
- You'll see it on homework 1

<http://groups.csail.mit.edu/pag/daikon>