

Dynamic Analysis

1

Small change in plans

- Today: we'll talk about dynamic analysis
 - Homework 1 is posted, due March 2
- Thursday: Guest lecture
- Next Tuesday: Guest lecture
- Next Thursday:
 - Finish up the overview of the class topics

2

Homework 1

- Posted (already) on class website
- Due Tuesday Mar 2, 9 AM on gradescope
- 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

3

Thursday, 2/16, noon in CS151

A Holistic View on Machine Learning for Systems

16 FEB Seminar
Speaker: Yi Ding

FREE


Add to Calendar

Thursday, 02/16/2023
12:00pm to 1:00pm

Computer Science Building
Room 150111

Abstract: Improving computer system performance and resource efficiency are long-standing goals. Recent approaches that use machine learning methods to achieve these goals rely on a predictor that predicts the latency, throughput, or energy consumption of a sub-computation to, for example, aid hardware resource management or scheduling.

In this talk, I will present a holistic view on machine learning for systems. I will demonstrate that the optimization goals between machine learning methods and systems problems do not always align, and this misalignment means that optimizing machine learning prediction accuracy does not optimize system behavior. Instead, my research vision focuses on a holistic view of machine learning for systems problems. The key steps in achieving this vision is making proper tradeoffs between different stages within the pipeline. Based on this vision, I will introduce a couple of machine learning for systems solutions to meet different systems goals including energy, performance, and interpretability. I will conclude the talk with my future directions.

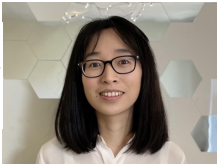


4

Tuesday, 2/21, noon in CS151

Trustworthy Software Enabled by Program Analysis and Synthesis

Abstract: Security, robustness, and fairness are all important non-functional properties of critical systems, such as software applications, microservices, servers, hardware, and finance. Unlike functional properties, which are easy to quantify or measure, these non-functional properties are difficult to guarantee. These non-functional properties are all security related. In this talk, I will present two techniques based on abstract interpreters and program synthesis for verifying security and fairness. The first one is a new method for detecting data-race bugs, which are a class of memory security issues that impact the stability of concurrent programs. The second one is the physical characteristics of the computing devices. Instead of hard-coding the entire algorithm and then trying to understand the hardware, the algorithm automatically synthesizes a hardware implementation. The second part of the talk is about the application of program analysis and synthesis in a business context, which is important for applications that are increasingly used to make socially sensitive decisions. Finally, I will talk about my research in the future, which will focus on providing formal guarantees of security, robustness, and fairness to other emerging applications.



Jingbo Wang

5

Any questions?

6

Today's plan

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

7

Rational Purify

- UNICOM
 - first Rational, then bought by IBM, then sold to UNICOM
- Memory debugging
 - uninitialized memory access
 - buffer overflow
 - improper freeing of memory
- Memory leak detection
 - memory blocks that no longer have a valid pointer

<https://teambue.unicomsi.com/products/purifyplus>

8

The Problem (for Purify to solve)

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

9

Memory



- What happens if we write here?

10

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

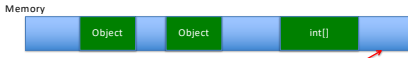
11

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

12

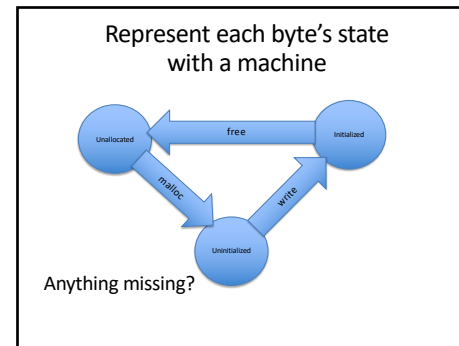
Memory



Memory

- What happens if we write here?

13



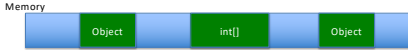
14

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

15

Memory



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.


16

Can we do better?

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

17

Padding between objects



Memory

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

18

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

19

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?

20

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.

21

Today's plan

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

22

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!

23

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.

24

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.

25

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 ≥ 0`

26

What is an invariant? (`j ≥ 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)`?

27

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 fails.

28

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`

29

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...

30

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

31

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.

32

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

33

Solution: Use statistics!

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

34

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 $0 \leq x, y < 1000$

$P(x == y) \approx .001$
 $P(x < y) \approx .5$
 $P(x != y) \approx .999$

35

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

36

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.

37

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?

38

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?

$\Theta(n^2)$

39

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

40

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!

41

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.

42

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) .

43

And others still

- We can do:

$$\begin{aligned} &\min(\text{array}) \\ &\max(\text{array}) \\ &\text{sum}(\text{array}) \\ &\text{etc.} \end{aligned}$$
- These expressions can be like variables:

$$x = \min(\text{array } z)$$

44

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

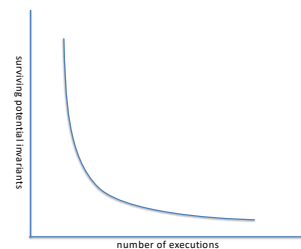
45

In practice

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

46

False invariants die quickly



47

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

<https://plse.cs.washington.edu/daikon/>

48