Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code

Elliot Greenlee and Jared M. Smith
CS560 Software Systems, April 13, 2017

Overview

- **Motivation**
- Terminology
- Internal Consistency
- Statistical Analyses
- Case Studies
- Limitations
- Review
- Discussion

So many bugs

- Human error
- Multiple developers
- Bad communication
- New conventions

- Manual Inspection
- Print statements
- Unit testing
- Debugging tools

Advanced bug finding

- Formal specification
- Hard coded checkers
- *Dynamic analyzers*
  - Daikon
  - Eraser

What is the obstacle?

- Already have many bugs
- Already have sophisticated techniques
- Already have constraints on correctness

- Knowing what rules to check
- Rules change between systems and programmers

Results

- Find rules by looking for contradictions in programming behavior
- Dramatically reduces manual effort
- Finds 10 to 100 times more contradiction instances
- Discovers contradictions that are hard to specify
- Doesn’t have to understand the system
- Found errors in Linux and OpenBSD that contributed to kernel patches

Overview

- Motivation
- **Terminology**
- Internal Consistency
- Statistical Analyses
- Case Studies
- Limitations
- Review
- Discussion

Beliefs

Beliefs are facts about the system implied by the code.

MUST Beliefs

- Beliefs that **are** true.

Dereferencing a pointer implies a belief that the pointer is not null

MAY Beliefs

- Beliefs that **could be** true.

Calling two functions one after the other like lock() and unlock() implies necessity

Templates

- *Templates*
  - Specification for rules
  - Ex. “<a> must be paired with <b>”
- *Slots*
  - Template positions that should be filled with concrete code elements
  - Ex. <a> and <b> are slots
- *Slot instances*
  - The code elements that fill slots
  - Ex. lock() == <a> and unlock() == <b>

Terminology

- *Code action*
  - An expression implying a belief
- *Propagate*
  - A belief set is propagated when it moves from one action to another
- *Static analysis*
  - In place before running the code
- *Dynamic analysis*
  - Checks paths as they are run

Overview

- Motivation
- Terminology
- **Internal Consistency**
- Statistical Analyses
- Case Studies
- Limitations
- Review
- Discussion

MUST Belief Analysis

- Look for contradictions
- Any contradiction implies an error

Consistency Checkers

1. **Template T**

   Determines what property the checker tests

2. **Valid slot instances for T**

   Each slot instance has a *belief set*

3. Belief implications from code actions

   Consider how the action affects the belief sets for each slot instance

4. Belief combination rules

   How do beliefs combine?

5. Belief propagation rules

   How do beliefs propagate?

Template rules

```
if (p == NULL) {
    std::cout << *p << std::endl;
} else {  }
```

Do not dereference a null pointer <x>

Consistency Checkers

1. Template T
   Determines what property the checker tests
2. Valid slot instances for T
   Each slot instance has a *belief set*
3. **Belief implications from code actions**
   Consider how the action affects the belief sets for each slot instance
4. Belief combination rules
   How do beliefs combine?
5. Belief propagation rules
   How do beliefs propagate?

What is the belief set for `p`?

```
if (p == NULL) {
    std::cout << *p << std::endl;
} else {  }
```

p is null

Consistency Checkers

1. Template T

   Determines what property the checker tests
2. Valid slot instances for T

   Each slot instance has a *belief set*
3. Belief implications from code actions

   Consider how the action affects the belief sets for each slot instance
4. **Belief combination rules**

   How do beliefs combine?
5. Belief propagation rules

   How do beliefs propagate?

How does the code action affect the belief set of every slot instance?

```
if (p == NULL) {
    std::cout << *p << std::endl;
} else {  }
```

The dereference of p implies that p is not null.

Consistency Checkers

1. Template T
   Determines what property the checker tests
2. Valid slot instances for T
   Each slot instance has a *belief set*
3. Belief implications from code actions
   Consider how the action affects the belief sets for each slot instance
4. Belief combination rules
   How do beliefs combine?
5. **Belief propagation rules**
   How do beliefs propagate?

How do beliefs propagate?

```
if (p == NULL) {
    std::cout << *p << std::endl;
} else {  }
```

`p == NULL` propagates the “null” belief to its true branch, the “not null” belief to its false branch, and either “null” or “not null” when the paths join.

Relating Code

- Code relationships help cross-check beliefs

- Implementation relation
  - An execution path from *a* to *b* means that *a*'s beliefs can be cross-checked with *b*'s beliefs
- Abstract relation
  - If *a* and *b* implement the same interface, they must assume the same execution context and fault model (example contradiction: *a* signals errors by returning positive integers, *b* returns negative integers)

Overview

- Motivation
- Terminology
- Internal Consistency
- **Statistical Analyses**
- Case Studies
- Limitations
- Review
- Discussion

MAY Belief Analysis

- Assume MAY Beliefs are MUST Beliefs
- Belief error ranking
  - A highly observed belief implies that the belief is correct and that errors are actually incorrect

Statistical Checker

1. Assume all combinations of beliefs are MUST beliefs
2. Counts the number of times a slot instance was checked and the number of times it failed the check (produced an error)
3. It ranks the errors from slot combinations from most to least plausible

Statistical Checker Example

```
lock l;  // Lock
int a, b; // Variables potentially protected by lock
void foo() {
  lock(l);
  a = a + b; // MAY: a, b protected by lock
  unlock(l);
  b = b + 1; // MUST: b not protected by lock
}
void bar() {
  lock(l);
  a = a + 1; // MAY: a protected by lock
  unlock(l);
}
void baz() {
  a = a + 1; // MAY: a protected by lock
  unlock(l);
  b = b - 1; // MUST: b not protected by lock
  a = a / b; // MUST: a not protected by lock
}
```

Statistical Checker example

- How many checks does `a` have?
    - 4
- How many errors does `a` have?
    - 1
- How many checks does `b` have?
    - 3
- How many errors does `b` have?
    - 2

- `a` produces an error 1 / 4 of the time
- `b` produces an error 2 / 3 of the time

Z Statistic for Proportions

- Sorts errors by rank to filter out implausible behavior
- Rank increases as sample population grows and number of counter-examples decreases
- Measures the number of standard errors away the observed ratio is from the expected ratio

Z Statistic for Proportions

- n is the population
- c is the number of counter-examples
- e is the number of successful checks
- p0 is the probability of successful checks
- 1-p0 is the probability of counter-examples

$$z(n, e) = (e/n - p_0)/\sqrt{(p_0 * (1 - p_0)/n}$$

Statistical Noise

- Coincidences and imperfect analysis leads to noise

- Noise can be countered with:
    - Large samples
    - Ranking error messages using z statistic value
    - Human-level operations

Latent Specifications

- Designed to communicate intent to other programmers
- Code must be human understandable
- Important operations coded explicitly
- Directly encoded in program
  - Naming convention
  - Special function
  - Data types
- Can be evaluated automatically because the consistency

Overview

- Motivation
- Terminology
- Internal Consistency
- Statistical Analyses
- **Case Studies**
- Limitations
- Review
- Discussion

Case Studies

- Internal Null Consistency
- A Security Checker
- Inferring Failure
- Deriving Temporal Rules

Internal Null Consistency

- Generalize pointer error discovery
- Finds three types of pointer errors
- do not dereference a null pointer <p>

  1. Check-then-use
  2. Use-then-check

- do not test a pointer <p> whose value is known

  3. Redundant checks

Internal Null Consistency

- Linux results

Checker	Bug	False
check-then-use	79	26
use-then-check	102	4
redundant-checks	24	10

A Security Checker

- Find kernel security errors
  - Operating systems cannot safely dereference pointers, so they use “paranoid” routines
  - Do not dereference user pointer <p>

1. Any pointer that is dereferenced is believed to be a safe kernel pointer
2. Any pointer that is sent to a "paranoid" routine is believed to be a "tainted" user pointer
3. Any pointer that is believed to be both a user pointer and a kernel pointer is an error

A Security Checker

- 35 security holes in Linux and OpenBSD
- Mostly driver code

OS	Errors	False	Applied
OpenBSD 2.8	18	3	1645
Linux 2.4.1	12 (3)	16 (1)	4905
Linux 2.3.99	5	n/a	n/a

A Security Checker

- Kernel backdoors to check if they were called from user or kernel code
- Largest source of false positives
- Highly stylized because of danger
- Susceptible to latent specification analysis

Inferring Failure

- Find errors where routines are not checked for failure or are incorrectly checked for failure
- Function <f> must be checked for failure
- Enormous number of these checks

Inferring Failure

- Two checkers written
    - Ensure that routines returning null pointers are checked before use
    - Ensure that routines that return integer error codes are checked

1. Assume all functions can fail
2. If the result of a function f is ignored or used without checks, emit error
3. If the result of a function is checked, emit a checked message.

Inferring Failure

- Linux and OpenBSD

Version	Bug	False
2.4.1	52 + 102	16
OpenBSD	27 + 14	21
Total	195	37

Deriving Temporal Rules

- Sequences of actions
  - No <a> after <b> (freed memory cannot be used)
  - <b> must follow <a> (unlock must follow lock)
  - In context <x>, do <b> after <a> (contextual rules)

- Implement the first two rules for specific cases
- Found instances of locking but never unlocking in the sound driver, and 23 total errors.

Deriving Temporal Rules

- No <a> after <b> deallocation
- Checks that freed memory is not used
- Many different deallocation functions
- If a function's argument is not used after the call, the programmer MAY believe it is a deallocation function
    - Assume all arguments are freed
    - For every function argument pair, count examples and errors
    - Rank with z-statistic

Deriving Temporal Rules

- <b> must follow <a>
- Blindly assume true for “plausible” pairs
  - Result of first function is used by a second
  - Variable is used in two sequential functions
  - Two no-argument function calls
- Count examples and counterexamples
- Rank errors using z-statistic

Overview

- Motivation
- Terminology
- Internal Consistency
- Statistical Analyses
- Case Studies
- **Limitations**
- Review
- Discussion

Overview

- Motivation
- Terminology
- Internal Consistency
- Statistical Analyses
- Case Studies
- Limitations
- **Review**
- Discussion

Review

- Internal consistency:
  - Finds contradictions in related code by combining must beliefs

- Statistical analyses
  - Compares examples and counterexamples for may beliefs, eventually establishing one or the other as correct based on consistency

Overview

- Motivation
- Terminology
- Internal Consistency
- Statistical Analyses
- Case Studies
- Limitations
- Review
- **Discussion**

Discussion Topics

- How would you improve belief accuracy?
- Have you seen bug detection in a compiler or IDE?
    - If not, should bug detection be a feature of compilers and IDEs?
- This method assumes correct coding practice. Is this assumption safe?